# Table of Contents

What is AWS Lambda? ............................................................................................................. 1
  When to use Lambda .............................................................................................................. 1
  Key features .......................................................................................................................... 2

Getting started .......................................................................................................................... 4
  Prerequisites ........................................................................................................................ 4
  Create a Lambda function with the console ........................................................................ 6
  Invoke the Lambda function using the console ................................................................. 12
  Clean up ............................................................................................................................... 14
  Additional resources and next steps ................................................................................. 15

Lambda foundations ............................................................................................................... 17
  Concepts ................................................................................................................................ 18
    Function ............................................................................................................................ 18
    Trigger .............................................................................................................................. 18
    Event ................................................................................................................................. 19
    Execution environment .................................................................................................... 19
    Instruction set architecture ............................................................................................. 20
    Deployment package ....................................................................................................... 20
    Runtime ............................................................................................................................. 20
    Layer ................................................................................................................................. 20
    Extension ........................................................................................................................... 21
    Concurrency ..................................................................................................................... 21
    Qualifier ............................................................................................................................ 22
    Destination ......................................................................................................................... 22
  Programming model ............................................................................................................. 23
  Execution environment ....................................................................................................... 25
    Runtime environment lifecycle ....................................................................................... 26
  Deployment packages .......................................................................................................... 32
    Container images ............................................................................................................. 32
    .zip file archives .............................................................................................................. 32
    Layers ............................................................................................................................... 34
  Using other AWS services ................................................................................................. 34

Infrastructure as code (IaC) ................................................................................................. 36
  IaC tools for Lambda .......................................................................................................... 36
  Getting started with IaC for Lambda .................................................................................. 38
Next steps .......................................................................................................................... 50
Supported regions for Lambda integration with Application Composer ....................... 50
Private networking ........................................................................................................ 52
VPC network elements .................................................................................................. 52
Connecting Lambda functions to your VPC .................................................................. 53
Shared subnets ............................................................................................................. 54
Lambda Hyperplane ENIs ............................................................................................. 54
Connections ................................................................................................................ 56
IPv6 support ................................................................................................................. 57
Security ......................................................................................................................... 58
Observability ............................................................................................................... 59
Lambda console ............................................................................................................. 60
Applications ................................................................................................................ 60
Functions ..................................................................................................................... 60
Code signing ................................................................................................................. 60
Layers ............................................................................................................................ 60
Edit code using the console editor ............................................................................... 61
Instruction sets (ARM/x86) ......................................................................................... 69
Advantages of using arm64 architecture .................................................................... 69
Requirements for migration to arm64 architecture .................................................... 70
Function code compatibility with arm64 architecture .................................................. 70
How to migrate to arm64 architecture ......................................................................... 70
Configuring the instruction set architecture ................................................................ 71
Additional features ..................................................................................................... 73
Scaling .......................................................................................................................... 73
Concurrency controls ................................................................................................... 73
Function URLs ............................................................................................................. 74
Asynchronous invocation ............................................................................................. 74
Event source mappings ............................................................................................... 75
Destinations ................................................................................................................ 76
Function blueprints ...................................................................................................... 77
Testing and deployment tools ...................................................................................... 77
Application templates ................................................................................................. 77
Learn how to build serverless solutions ....................................................................... 78
Lambda runtimes ......................................................................................................... 79
New runtime releases ................................................................................................. 82
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Future runtime launch dates</td>
<td>83</td>
</tr>
<tr>
<td>Runtime deprecation policy</td>
<td>83</td>
</tr>
<tr>
<td>Runtime support after deprecation</td>
<td>84</td>
</tr>
<tr>
<td>Receiving runtime deprecation notifications</td>
<td>84</td>
</tr>
<tr>
<td>Listing affected functions using the AWS CLI</td>
<td>85</td>
</tr>
<tr>
<td>Deprecated runtimes</td>
<td>86</td>
</tr>
<tr>
<td>Runtime updates</td>
<td>89</td>
</tr>
<tr>
<td>Runtime management controls</td>
<td>90</td>
</tr>
<tr>
<td>Two-phase runtime version rollout</td>
<td>91</td>
</tr>
<tr>
<td>Roll back a runtime version</td>
<td>91</td>
</tr>
<tr>
<td>Identifying runtime version changes</td>
<td>93</td>
</tr>
<tr>
<td>Configure runtime management settings</td>
<td>95</td>
</tr>
<tr>
<td>Shared responsibility model</td>
<td>96</td>
</tr>
<tr>
<td>High-compliance applications</td>
<td>97</td>
</tr>
<tr>
<td>Runtime modifications</td>
<td>99</td>
</tr>
<tr>
<td>Language-specific environment variables</td>
<td>99</td>
</tr>
<tr>
<td>Wrapper scripts</td>
<td>99</td>
</tr>
<tr>
<td>Runtime API</td>
<td>103</td>
</tr>
<tr>
<td>Next invocation</td>
<td>103</td>
</tr>
<tr>
<td>Invocation response</td>
<td>105</td>
</tr>
<tr>
<td>Initialization error</td>
<td>105</td>
</tr>
<tr>
<td>Invocation error</td>
<td>107</td>
</tr>
<tr>
<td>OS-only runtimes</td>
<td>109</td>
</tr>
<tr>
<td>Building a custom runtime</td>
<td>111</td>
</tr>
<tr>
<td>Requirements</td>
<td>111</td>
</tr>
<tr>
<td>Implementing response streaming in a custom runtime</td>
<td>113</td>
</tr>
<tr>
<td>Custom runtime tutorial</td>
<td>115</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>115</td>
</tr>
<tr>
<td>Create a function</td>
<td>116</td>
</tr>
<tr>
<td>Create a layer</td>
<td>119</td>
</tr>
<tr>
<td>Update the function</td>
<td>119</td>
</tr>
<tr>
<td>Update the runtime</td>
<td>121</td>
</tr>
<tr>
<td>Share the layer</td>
<td>122</td>
</tr>
<tr>
<td>Clean up</td>
<td>122</td>
</tr>
<tr>
<td>AVX2 vectorization</td>
<td>124</td>
</tr>
<tr>
<td>Compiling from source</td>
<td>124</td>
</tr>
</tbody>
</table>
Enabling AVX2 for Intel MKL ................................................................. 125
AVX2 support in other languages ....................................................... 125

**Configuring functions** .............................................................................. 127

Configuring function options ..................................................................... 129
  Function versions .................................................................................. 129
  Using the function overview ................................................................. 130
Configuring functions (console) ................................................................. 130
Configuring function memory (console) ................................................... 132
Accepting function memory recommendations (console) .......................... 132
Configuring function timeout (console) .................................................... 132
Configuring ephemeral storage (console) ................................................. 132
Configuring triggers (console) ................................................................. 133
Configuring Lambda advanced logging options ...................................... 133
Connecting RDS databases (console) ....................................................... 133

Environment variables .............................................................................. 134
  Configuring environment variables ...................................................... 134
  Configuring environment variables with the API ................................... 136
  Example scenario for environment variables ....................................... 137
Retrieve environment variables ............................................................... 137
Defined runtime environment variables .................................................. 139
Securing environment variables ............................................................. 141
Sample code and templates ................................................................. 144

Outbound networking .............................................................................. 145
  Managing VPC connections .................................................................. 146
Execution role and user permissions ....................................................... 146
Configuring VPC access (console) .......................................................... 147
Configuring VPC access (API) ................................................................. 148
Using IAM condition keys for VPC settings ............................................ 149
Internet and service access for VPC-connected functions ......................... 154
VPC tutorials ......................................................................................... 154
Sample VPC configurations .................................................................. 155

Inbound networking .............................................................................. 156
  Considerations for Lambda interface endpoints .................................... 156
Creating an interface endpoint for Lambda .............................................. 157
Creating an interface endpoint policy for Lambda ................................... 158
File system .............................................................................................. 160
Execution role and user permissions ................................................................. 160
Configuring a file system and access point ..................................................... 161
Connecting to a file system (console) ............................................................... 162
Configuring file system access with the Lambda API ....................................... 163
Mounting an Amazon EFS file system in another AWS account ....................... 164
AWS CloudFormation and AWS SAM .............................................................. 165
Sample applications ......................................................................................... 167
Aliases .................................................................................................................. 168
Creating a function alias (Console) ................................................................. 168
Managing aliases with the Lambda API ............................................................ 169
Managing aliases with AWS SAM and AWS CloudFormation ......................... 169
Using aliases ...................................................................................................... 169
Resource policies ............................................................................................... 170
Alias routing configuration .................................................................................. 170
Versions .............................................................................................................. 174
Creating function versions ............................................................................... 174
Using versions .................................................................................................... 176
Granting permissions ........................................................................................ 176
Response streaming ........................................................................................... 177
Writing response streaming-enabled functions ............................................... 177
Invoking a response streaming enabled function using Lambda function URLs .... 179
Bandwidth limits for response streaming .......................................................... 180
Tutorial: Creating a response streaming function with a function URL ............ 181

Deploying functions ........................................................................................... 187

.zip file archives .............................................................................................. 187
Container images ............................................................................................. 187

.zip file archives .............................................................................................. 189

Creating the function ....................................................................................... 189
Using the console code editor .......................................................................... 191
Updating function code ................................................................................... 191
Changing the runtime ...................................................................................... 192
Changing the architecture ............................................................................... 192
Using the Lambda API .................................................................................... 193
AWS CloudFormation ....................................................................................... 193
Container images ............................................................................................. 194
Requirements .................................................................................................... 195
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using an AWS base image</td>
<td>196</td>
</tr>
<tr>
<td>Using an AWS OS-only base image</td>
<td>196</td>
</tr>
<tr>
<td>Using a non-AWS base image</td>
<td>197</td>
</tr>
<tr>
<td>Runtime interface clients</td>
<td>198</td>
</tr>
<tr>
<td>Amazon ECR permissions</td>
<td>198</td>
</tr>
<tr>
<td>Container image settings</td>
<td>201</td>
</tr>
<tr>
<td>Testing images</td>
<td>203</td>
</tr>
<tr>
<td>Guidelines</td>
<td>203</td>
</tr>
<tr>
<td>Environment variables</td>
<td>203</td>
</tr>
<tr>
<td>Testing AWS base images</td>
<td>204</td>
</tr>
<tr>
<td>Testing OS-only base images</td>
<td>206</td>
</tr>
<tr>
<td>Testing non-AWS images</td>
<td>208</td>
</tr>
<tr>
<td><strong>Invoking functions</strong></td>
<td><strong>215</strong></td>
</tr>
<tr>
<td>Testing in console</td>
<td>217</td>
</tr>
<tr>
<td>Invoking functions with test events</td>
<td>217</td>
</tr>
<tr>
<td>Creating private test events</td>
<td>218</td>
</tr>
<tr>
<td>Creating shareable test events</td>
<td>218</td>
</tr>
<tr>
<td>Deleting shareable test event schemas</td>
<td>220</td>
</tr>
<tr>
<td>Synchronous invocation</td>
<td>221</td>
</tr>
<tr>
<td>Asynchronous invocation</td>
<td>225</td>
</tr>
<tr>
<td>How Lambda handles asynchronous invocations</td>
<td>225</td>
</tr>
<tr>
<td>Configuring error handling for asynchronous invocation</td>
<td>228</td>
</tr>
<tr>
<td>Configuring destinations for asynchronous invocation</td>
<td>228</td>
</tr>
<tr>
<td>Asynchronous invocation configuration API</td>
<td>233</td>
</tr>
<tr>
<td>Dead-letter queues</td>
<td>234</td>
</tr>
<tr>
<td>Event source mapping</td>
<td>238</td>
</tr>
<tr>
<td>Batching behavior</td>
<td>240</td>
</tr>
<tr>
<td>Configuring destinations for event source mapping invocations</td>
<td>245</td>
</tr>
<tr>
<td>Event filtering</td>
<td>248</td>
</tr>
<tr>
<td>Event filtering basics</td>
<td>249</td>
</tr>
<tr>
<td>Handling records that don't meet filter criteria</td>
<td>251</td>
</tr>
<tr>
<td>Filter rule syntax</td>
<td>252</td>
</tr>
<tr>
<td>Attaching filter criteria to an event source mapping (console)</td>
<td>253</td>
</tr>
<tr>
<td>Attaching filter criteria to an event source mapping (AWS CLI)</td>
<td>254</td>
</tr>
<tr>
<td>Attaching filter criteria to an event source mapping (AWS SAM)</td>
<td>255</td>
</tr>
<tr>
<td>Using filters with different AWS services</td>
<td>256</td>
</tr>
</tbody>
</table>
Filtering with DynamoDB ................................................................. 257
Filtering with Kinesis ................................................................. 264
Filtering with Amazon MQ .......................................................... 268
Filtering with Amazon MSK and self-managed Apache Kafka ................. 274
Filtering with Amazon SQS .......................................................... 278
Function states .................................................................................. 284
  Function states while updating ....................................................... 285
Error handling .................................................................................. 287
Invoking functions defined as container images ..................................... 290
  Function lifecycle ......................................................................... 290
  Invoking the function ................................................................... 290
  Image security ............................................................................... 291
Recursive loop detection ..................................................................... 292
  Understanding recursive loop detection ......................................... 292
Supported AWS services and SDKs .................................................... 294
Recursive loop notifications ............................................................. 296
Responding to recursive loop detection notifications ............................. 297
Function URLs .................................................................................. 299
  Creating and managing function URLs ........................................... 301
  Security and auth model ............................................................... 309
  Invoking function URLs ............................................................... 317
  Monitoring function URLs ............................................................ 328
Tutorial: Creating a function with a function URL ................................. 330
Managing functions .......................................................................... 335
Tutorial - Lambda with CLI ............................................................. 336
  Prerequisites ................................................................................ 336
  Create the execution role .............................................................. 337
  Create the function ...................................................................... 338
  Update the function ..................................................................... 342
  List the Lambda functions in your account ..................................... 342
  Retrieve a Lambda function .......................................................... 343
  Clean up ....................................................................................... 344
Function scaling ................................................................................. 345
  Understanding and visualizing concurrency .................................... 345
  How to calculate concurrency ....................................................... 350
  Concurrency vs. requests per second .............................................. 351
Reserved concurrency and provisioned concurrency .................................................. 352
Concurrency quotas ................................................................................................. 362
Configuring reserved concurrency ........................................................................ 364
Configuring provisioned concurrency ................................................................... 368
Scaling behavior ...................................................................................................... 378
Monitoring concurrency ......................................................................................... 379
Code signing ........................................................................................................... 385
Signature validation .................................................................................................. 386
Configuration prerequisites ...................................................................................... 387
Creating code signing configurations ....................................................................... 387
Updating a code signing configuration ..................................................................... 387
Deleting a code signing configuration ....................................................................... 388
Enabling code signing for a function ........................................................................ 388
Configuring IAM policies ......................................................................................... 389
Configuring code signing with the Lambda API ..................................................... 390
Tags ......................................................................................................................... 391
Permissions ............................................................................................................... 391
Using tags with the console ...................................................................................... 391
Using tags with the AWS CLI ................................................................................... 394
Requirements for tags ............................................................................................... 395
Testing functions ...................................................................................................... 396
Targeted business outcomes ..................................................................................... 397
What to test ............................................................................................................... 397
How to test serverless ............................................................................................... 398
Testing techniques ..................................................................................................... 399
  Testing in the cloud .................................................................................................. 399
  Testing with mocks .................................................................................................. 402
  Testing with emulation ............................................................................................ 403
Best practices ............................................................................................................ 404
  Prioritize testing in the cloud ............................................................................... 404
  Structure your code for testability ......................................................................... 404
  Accelerate development feedback loops ............................................................... 405
  Focus on integration tests ...................................................................................... 405
  Create isolated test environments ......................................................................... 406
  Use mocks for isolated business logic .................................................................... 407
  Use emulators sparingly ......................................................................................... 407
Building with Node.js ................................................................. 412

Node.js initialization ...................................................................... 414
  Designating a function handler as an ES module .......................... 415
Using keep-alive ........................................................................... 415
CA certificate loading ..................................................................... 416
Handler .......................................................................................... 418
  Using async/await .................................................................. 419
  Using callbacks ................................................................... 421
Deploy .zip file archives ................................................................. 425
  Runtime dependencies in Node.js ............................................. 425
Creating a .zip deployment package with no dependencies .......... 426
Creating a .zip deployment package with dependencies ............. 426
Creating a Node.js layer for your dependencies .......................... 427
Dependency search path and runtime-included libraries .............. 428
Creating and updating Node.js Lambda functions using .zip files 429
Deploy container images ............................................................... 436
  AWS base images for Node.js .................................................. 437
Using an AWS base image .......................................................... 437
Using a non-AWS base image ..................................................... 443
Context ......................................................................................... 453
Logging ........................................................................................ 455
  Creating a function that returns logs ....................................... 455
  Using Lambda advanced logging controls with Node.js .......... 457
Using the Lambda console .......................................................... 463
Using the CloudWatch console .................................................... 463
Using the AWS Command Line Interface (AWS CLI) .................. 464
Deleting logs ............................................................................... 467
Errors ............................................................................................. 468
  Syntax ......................................................................................... 468
  How it works ............................................................................ 469
  Using the Lambda console ..................................................... 470
Using the AWS Command Line Interface (AWS CLI) .............................................. 470
Error handling in other AWS services ................................................................. 471
What's next? ............................................................................................................. 472
Tracing ..................................................................................................................... 473
  Using ADOT to instrument your Node.js functions ........................................... 474
  Using the X-Ray SDK to instrument your Node.js functions ......................... 474
Activating tracing with the Lambda console ....................................................... 475
Activating tracing with the Lambda API ............................................................. 476
Activating tracing with AWS CloudFormation .................................................. 476
Interpreting an X-Ray trace .................................................................................. 477
Storing runtime dependencies in a layer (X-Ray SDK) ...................................... 479

Building with TypeScript ...................................................................................... 481
Development environment ..................................................................................... 481
Handler .................................................................................................................. 484
  Using async/await ............................................................................................... 485
  Using callbacks .................................................................................................. 486
  Using types for the event object ...................................................................... 487
Deploy .zip file archives ....................................................................................... 489
  Using AWS SAM ............................................................................................... 489
  Using the AWS CDK ......................................................................................... 491
  Using the AWS CLI and esbuild ...................................................................... 494
Deploy container images ....................................................................................... 497
  Using a Node.js base image to build and package TypeScript function code ...... 497
Context ................................................................................................................ 504
Logging ................................................................................................................... 506
  Tools and libraries ............................................................................................. 506
  Using Powertools for AWS Lambda (TypeScript) and AWS SAM for structured logging ..... 507
  Using Powertools for AWS Lambda (TypeScript) and the AWS CDK for structured logging ................................................................. 509
  Using the Lambda console .............................................................................. 513
  Using the CloudWatch console .................................................................... 513
Testing ................................................................................................................... 515
  Testing your serverless applications ............................................................... 516
Errors ..................................................................................................................... 518
Tracing ................................................................................................................... 521
  Using Powertools for AWS Lambda (TypeScript) and AWS SAM for tracing .......... 522
| Using Powertools for AWS Lambda (TypeScript) and the AWS CDK for tracing | 524 |
| Interpreting an X-Ray trace | 528 |
| **Building with Python** | 529 |
| Response format | 531 |
| Graceful shutdown for extensions | 532 |
| Handler | 533 |
| Naming | 533 |
| How it works | 534 |
| Returning a value | 534 |
| Examples | 535 |
| Deploy .zip file archives | 538 |
| Runtime dependencies in Python | 538 |
| Creating a .zip deployment package with no dependencies | 539 |
| Creating a .zip deployment package with dependencies | 540 |
| Creating a Python layer for your dependencies | 542 |
| Dependency search path and runtime-included libraries | 543 |
| Using __pycache__ folders | 544 |
| Creating .zip deployment packages with native libraries | 545 |
| Creating and updating Python Lambda functions using .zip files | 546 |
| Deploy container images | 553 |
| AWS base images for Python | 554 |
| Using an AWS base image | 555 |
| Using a non-AWS base image | 561 |
| Context | 570 |
| Logging | 572 |
| Printing to the log | 572 |
| Using a logging library | 573 |
| Using Lambda advanced logging controls with Python | 575 |
| Viewing logs in Lambda console | 579 |
| Viewing logs in CloudWatch console | 580 |
| Viewing logs with AWS CLI | 580 |
| Deleting logs | 583 |
| Tools and libraries | 583 |
| Using Powertools for AWS Lambda (Python) and AWS SAM for structured logging | 584 |
| Using Powertools for AWS Lambda (Python) and AWS CDK for structured logging | 588 |
| Testing | 595 |
Using the Lambda console ........................................................................................................... 655
Using the CloudWatch console ................................................................................................. 656
Using the AWS Command Line Interface (AWS CLI) ............................................................... 656
Deleting logs .............................................................................................................................. 659
Logger library .......................................................................................................................... 660
Errors ........................................................................................................................................ 661
Syntax ....................................................................................................................................... 661
How it works .............................................................................................................................. 662
Using the Lambda console ....................................................................................................... 663
Using the AWS Command Line Interface (AWS CLI) .............................................................. 663
Error handling in other AWS services ...................................................................................... 664
Sample applications ................................................................................................................. 665
What’s next? .............................................................................................................................. 665

Tracing ....................................................................................................................................... 666
Enabling active tracing with the Lambda API ........................................................................... 670
Enabling active tracing with AWS CloudFormation ............................................................... 671
Storing runtime dependencies in a layer ................................................................................... 671

Building with Java .................................................................................................................... 673

Handler ..................................................................................................................................... 676
Example handler: Java 17 runtimes ......................................................................................... 676
Example handler: Java 11 runtimes and below ......................................................................... 678
Initialization code ..................................................................................................................... 679
Choosing input and output types .............................................................................................. 680
Handler interfaces .................................................................................................................... 681
Sample handler code ................................................................................................................ 683
Deploy .zip file archives .......................................................................................................... 684
Prerequisites .............................................................................................................................. 684
Tools and libraries ..................................................................................................................... 684
Building a deployment package with Gradle .......................................................................... 686
Creating a Java layer for your dependencies ......................................................................... 687
Building a deployment package with Maven ........................................................................... 688
Uploading a deployment package with the Lambda console .................................................. 690
Uploading a deployment package with the AWS CLI ............................................................... 692
Uploading a deployment package with AWS SAM ................................................................. 693
Deploy container images .......................................................................................................... 696
AWS base images for Java ........................................................................................................ 697
Using an AWS base image ........................................................................................................... 697
Using a non-AWS base image ................................................................................................. 706
Java customization .................................................................................................................. 717
  JAVA_TOOL_OPTIONS environment variable ...................................................................... 717
Context ..................................................................................................................................... 720
  Context in sample applications ............................................................................................ 722
Logging .................................................................................................................................... 724
  Creating a function that returns logs .................................................................................... 724
  Using Lambda advanced logging controls with Java ........................................................... 726
Advanced logging with Log4j2 and SLF4J ............................................................................. 729
Tools and libraries .................................................................................................................. 732
  Using Powertools for AWS Lambda (Java) and AWS SAM for structured logging ............ 733
Using the Lambda console ...................................................................................................... 737
Using the CloudWatch console .............................................................................................. 737
Using the AWS Command Line Interface (AWS CLI) ............................................................ 738
Deleting logs ........................................................................................................................... 741
Sample logging code .............................................................................................................. 741
Errors ....................................................................................................................................... 743
  Syntax ................................................................................................................................. 743
  How it works ........................................................................................................................ 744
  Creating a function that returns exceptions ........................................................................ 745
Using the Lambda console ...................................................................................................... 746
Using the AWS Command Line Interface (AWS CLI) ............................................................ 747
Error handling in other AWS services ..................................................................................... 748
Sample applications ............................................................................................................... 749
What's next? ............................................................................................................................ 750
Tracing ....................................................................................................................................... 751
  Using Powertools for AWS Lambda (Java) and AWS SAM for tracing ............................... 752
  Using Powertools for AWS Lambda (Java) and the AWS CDK for tracing ...................... 754
  Using ADOT to instrument your Java functions ................................................................ 766
  Using the X-Ray SDK to instrument your Java functions ................................................. 766
Activating tracing with the Lambda console ......................................................................... 767
Activating tracing with the Lambda API ................................................................................ 767
Activating tracing with AWS CloudFormation ..................................................................... 768
Interpreting an X-Ray trace .................................................................................................... 768
Storing runtime dependencies in a layer (X-Ray SDK) ........................................................ 771
# AWS Lambda

**X-Ray tracing in sample applications (X-Ray SDK)** .......................................................... 772
**Tutorial - Eclipse IDE** .......................................................................................................... 773
  **Prerequisites** ..................................................................................................................... 773
  **Create and build a project** .................................................................................................. 773
**Sample apps** ....................................................................................................................... 777

## Building with Go ..................................................................................................................... 779
**Go runtime support** ............................................................................................................. 779
**Tools and libraries** ............................................................................................................... 780
**Handler** .................................................................................................................................. 782
  **Naming** .............................................................................................................................. 784
  **Lambda function handler using structured types** ............................................................. 784
  **Using global state** ............................................................................................................. 786
**Context** .................................................................................................................................. 789
  **Accessing invoke context information** ............................................................................. 789
**Deploy .zip file archives** ...................................................................................................... 792
  **Creating a .zip file on macOS and Linux** ......................................................................... 792
  **Creating a .zip file on Windows** ....................................................................................... 795
  **Creating and updating Go Lambda functions using .zip files** ........................................ 799
  **Creating a Go layer for your dependencies** ...................................................................... 805
**Deploy container images** ...................................................................................................... 806
  **AWS base images for Go** ............................................................................................... 806
  **Go runtime interface client** ............................................................................................. 807
  **Using an AWS OS-only base image** .................................................................................. 807
  **Using a non-AWS base image** .......................................................................................... 814
  **Using the Go 1.x base image** ........................................................................................... 821
**Logging** .................................................................................................................................. 829
  **Creating a function that returns logs** .............................................................................. 829
  **Using the Lambda console** ............................................................................................ 831
  **Using the CloudWatch console** ...................................................................................... 831
  **Using the AWS Command Line Interface (AWS CLI)** ..................................................... 831
  **Deleting logs** ................................................................................................................... 835
**Errors** ................................................................................................................................... 836
  **Creating a function that returns exceptions** ................................................................. 836
  **How it works** .................................................................................................................... 837
  **Using the Lambda console** ............................................................................................ 838
  **Using the AWS Command Line Interface (AWS CLI)** ..................................................... 838
Error handling in other AWS services ................................................................. 839
What’s next? ........................................................................................................... 840
Tracing ................................................................................................................... 841
Using ADOT to instrument your Go functions .................................................... 842
Using the X-Ray SDK to instrument your Go functions ....................................... 842
Activating tracing with the Lambda console ...................................................... 842
Activating tracing with the Lambda API ............................................................ 843
Activating tracing with AWS CloudFormation .................................................. 843
Interpreting an X-Ray trace ................................................................................ 844
Environment variables ......................................................................................... 847

**Building with C#** ............................................................................................. 848

Development environment .................................................................................. 850
Handler .................................................................................................................. 851
  .NET execution models for Lambda ................................................................. 851
  Class library handlers ....................................................................................... 852
  Executable assembly handlers ......................................................................... 853
  Serialization in Lambda functions .................................................................. 854
  Simplify function code with the Lambda Annotations framework .................. 856
Lambda function handler restrictions .................................................................. 859
Deployment package ............................................................................................ 860
  Using the .NET Lambda Global CLI ............................................................... 861
  Using the AWS Serverless Application Model (AWS SAM) ............................ 867
  Using the AWS Cloud Development Kit (AWS CDK) ...................................... 870
  Deploy ASP.NET applications ......................................................................... 874
Deploy container images ....................................................................................... 879
  AWS base images for .NET ............................................................................. 880
  Using an AWS base image ............................................................................. 880
  Using a non-AWS base image ....................................................................... 883
Native AOT compilation ....................................................................................... 887
  Lambda runtime .............................................................................................. 887
  Prerequisites ..................................................................................................... 887
Getting started ...................................................................................................... 888
Serialization .......................................................................................................... 890
Trimming ............................................................................................................... 891
Troubleshooting ................................................................................................... 891
Context .................................................................................................................. 892
Logging .................................................................................................................. 894
  Creating a function that returns logs ................................................................. 894
Tools and libraries ................................................................................................. 895
Using Powertools for AWS Lambda (.NET) and AWS SAM for structured logging ............................................................................. 895
Using the Lambda console ...................................................................................... 898
Using the CloudWatch console .............................................................................. 898
Using the AWS Command Line Interface (AWS CLI) .......................................... 899
Deleting logs .......................................................................................................... 902
Errors .................................................................................................................... 903
  Syntax .................................................................................................................. 903
  How it works ........................................................................................................ 906
  Using the Lambda console ................................................................................... 907
  Using the AWS Command Line Interface (AWS CLI) ....................................... 908
  Error handling in other AWS services ............................................................... 909
  What’s next? ....................................................................................................... 910
Tracing ................................................................................................................... 911
  Using Powertools for AWS Lambda (.NET) and AWS SAM for tracing .......... 912
  Using the X-Ray SDK to instrument your .NET functions ................................ 915
  Activating tracing with the Lambda console .................................................... 916
  Activating tracing with the Lambda API ............................................................ 917
  Activating tracing with AWS CloudFormation ............................................... 917
  Interpreting an X-Ray trace ................................................................................ 918
Testing .................................................................................................................... 921
  Testing your serverless applications .................................................................. 922
Building with PowerShell ....................................................................................... 925
  Development Environment ................................................................................. 927
  Deployment package ........................................................................................... 928
    Creating a Lambda function .............................................................................. 928
Handler .................................................................................................................. 930
  Returning data .................................................................................................... 931
Context .................................................................................................................. 932
Logging .................................................................................................................. 933
  Creating a function that returns logs ................................................................. 933
  Using the Lambda console ................................................................................... 935
  Using the CloudWatch console ......................................................................... 935
  Using the AWS Command Line Interface (AWS CLI) ....................................... 935
Building with Rust ................................................................. 945
  Handler ............................................................................... 947
    Using shared state ............................................................. 948
  Context ............................................................................... 950
    Accessing invoke context information ................................ 950
  HTTP events ......................................................................... 952
  Deploy .zip file archives ....................................................... 955
    Prerequisites ....................................................................... 955
  Building the function ............................................................ 955
  Deploying the function ............................................................ 956
  Invoking the function ............................................................. 958
  Logging ................................................................................. 959
    Creating a function that writes logs ...................................... 959
    Advanced logging with the Tracing crate .............................. 959
  Errors .................................................................................. 962
    Creating a function that returns errors ................................. 962
Integrating other services ......................................................... 963
  Listing of services and links to more information ..................... 963
  Event-driven invocation ......................................................... 966
  Lambda polling ..................................................................... 966
  Use cases .............................................................................. 968
    Example 1: Amazon S3 pushes events and invokes a Lambda function .............................................. 969
    Example 2: AWS Lambda pulls events from a Kinesis stream and invokes a Lambda function .................. 969
  Alexa ..................................................................................... 971
  API Gateway ......................................................................... 972
    Adding an endpoint to your Lambda function ....................... 972
    Proxy integration .............................................................. 973
Example event ......................................................................................................................... 1077
Polling and batching streams ............................................................................................. 1078
Polling and stream starting positions ................................................................................ 1079
Simultaneous readers .......................................................................................................... 1080
Execution role permissions ............................................................................................... 1080
Configuring a stream as an event source .......................................................................... 1080
Event source mapping APIs .............................................................................................. 1082
Error handling .................................................................................................................... 1084
Amazon CloudWatch metrics ............................................................................................ 1086
Time windows ...................................................................................................................... 1086
Reporting batch item failures .......................................................................................... 1091
Amazon DynamoDB Streams configuration parameters .................................................. 1094
Tutorial .................................................................................................................................. 1096
Sample code ........................................................................................................................ 1104
Sample template ................................................................................................................ 1108
EC2 ........................................................................................................................................ 1110
Permissions .......................................................................................................................... 1111
ElastiCache ............................................................................................................................ 1112
Application Load Balancer ................................................................................................. 1113
EFS ......................................................................................................................................... 1115
  Connections ........................................................................................................................ 1116
  Throughput .......................................................................................................................... 1116
  IOPS .................................................................................................................................... 1117
EventBridge Scheduler ......................................................................................................... 1118
  Set up the execution role .................................................................................................. 1118
  Create a schedule .............................................................................................................. 1118
  Related resources ............................................................................................................. 1123
IoT ......................................................................................................................................... 1124
IoT Events ............................................................................................................................. 1126
Apache Kafka ........................................................................................................................ 1128
  Example event .................................................................................................................... 1129
  Kafka cluster authentication ............................................................................................ 1130
  Managing API access and permissions ......................................................................... 1133
  Authentication and authorization errors .......................................................................... 1136
  Network configuration ..................................................................................................... 1138
  Adding a Kafka cluster as an event source ..................................................................... 1139
Event source mapping errors ............................................................... 1214
Amazon MQ and RabbitMQ configuration parameters ......................... 1214
MSK .................................................................................................. 1216
Example event .................................................................................. 1217
MSK cluster authentication .................................................................. 1218
Managing API access and permissions ............................................... 1222
Authentication and authorization errors .............................................. 1226
Network configuration ......................................................................... 1228
Adding Amazon MSK as an event source ......................................... 1229
Cross-account event source mappings ............................................... 1237
Auto scaling of the Amazon MSK event source ................................. 1238
Polling and stream starting positions .................................................. 1239
Amazon CloudWatch metrics ............................................................. 1239
Amazon MSK configuration parameters ............................................. 1240
RDS .................................................................................................. 1242
Configuring your function .................................................................. 1242
Process event notifications from Amazon RDS .................................... 1243
Lambda and Amazon RDS tutorial ....................................................... 1244
S3 ...................................................................................................... 1245
Tutorial: Use an S3 trigger .................................................................. 1246
Tutorial: Use an Amazon S3 trigger to create thumbnails .................... 1269
S3 Batch ............................................................................................ 1305
Invoking Lambda functions from Amazon S3 batch operations .......... 1306
S3 Object Lambda .............................................................................. 1308
Secrets Manager ................................................................................ 1309
SES ..................................................................................................... 1310
SNS ..................................................................................................... 1313
Tutorial ............................................................................................... 1314
Sample code ...................................................................................... 1330
SQS .................................................................................................... 1333
Example standard queue message event .......................................... 1333
Example FIFO queue message event ............................................... 1335
Configuring a queue to use with Lambda .......................................... 1336
Execution role permissions ................................................................. 1336
Configuring a queue as an event source ........................................... 1337
Scaling and processing ..................................................................... 1339
## Maximum concurrency

- Event source mapping APIs
- Backoff strategy for failed invocations
- Implementing partial batch responses
- Amazon SQS configuration parameters
- Tutorial

## AWS Lambda Function configuration

- AWS managed policies for Lambda features
- VPC Lattice concepts
- Sample template
- Sample code
- Amazon SQS configuration parameters

## Best practices

- Function code
- Function configuration
- Metrics and alarms
- Working with streams
- Security best practices

## Access permissions

- Execution role
  - Creating an execution role in the IAM console
  - Grant least privilege access to your Lambda execution role
  - Managing roles with the IAM API
  - Session duration for temporary security credentials
  - AWS managed policies for Lambda features
  - Working with Lambda execution environment credentials
- User policies
- Function development
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer development and use</td>
<td>1415</td>
</tr>
<tr>
<td>Cross-account roles</td>
<td>1416</td>
</tr>
<tr>
<td>Condition keys for VPC settings</td>
<td>1416</td>
</tr>
<tr>
<td>Control access using tags</td>
<td>1417</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>1418</td>
</tr>
<tr>
<td>Step 1: Require tags</td>
<td>1418</td>
</tr>
<tr>
<td>Step 2: Control actions using tags</td>
<td>1419</td>
</tr>
<tr>
<td>Step 3: Grant list permissions</td>
<td>1419</td>
</tr>
<tr>
<td>Step 4: Grant IAM permissions</td>
<td>1420</td>
</tr>
<tr>
<td>Step 5: Create the IAM role</td>
<td>1421</td>
</tr>
<tr>
<td>Step 6: Create the IAM user</td>
<td>1421</td>
</tr>
<tr>
<td>Step 7: Test the permissions</td>
<td>1421</td>
</tr>
<tr>
<td>Resource-based policies</td>
<td>1423</td>
</tr>
<tr>
<td>Supported API actions</td>
<td>1425</td>
</tr>
<tr>
<td>Granting function access to AWS services</td>
<td>1426</td>
</tr>
<tr>
<td>Granting function access to an organization</td>
<td>1427</td>
</tr>
<tr>
<td>Granting function access to other accounts</td>
<td>1428</td>
</tr>
<tr>
<td>Granting layer access to other accounts</td>
<td>1429</td>
</tr>
<tr>
<td>Cleaning up resource-based policies</td>
<td>1430</td>
</tr>
<tr>
<td>Resources and conditions</td>
<td>1432</td>
</tr>
<tr>
<td>Policy conditions</td>
<td>1433</td>
</tr>
<tr>
<td>Function resource names</td>
<td>1434</td>
</tr>
<tr>
<td>Function actions</td>
<td>1436</td>
</tr>
<tr>
<td>Event source mapping actions</td>
<td>1440</td>
</tr>
<tr>
<td>Layer actions</td>
<td>1440</td>
</tr>
<tr>
<td>Permissions boundaries</td>
<td>1442</td>
</tr>
<tr>
<td>Security, governance, and compliance</td>
<td>1445</td>
</tr>
<tr>
<td>Data protection</td>
<td>1446</td>
</tr>
<tr>
<td>Encryption in transit</td>
<td>1447</td>
</tr>
<tr>
<td>Encryption at rest</td>
<td>1447</td>
</tr>
<tr>
<td>Identity and access management</td>
<td>1447</td>
</tr>
<tr>
<td>Audience</td>
<td>1448</td>
</tr>
<tr>
<td>Authenticating with identities</td>
<td>1448</td>
</tr>
<tr>
<td>Managing access using policies</td>
<td>1451</td>
</tr>
<tr>
<td>How AWS Lambda works with IAM</td>
<td>1454</td>
</tr>
<tr>
<td>Identity-based policy examples</td>
<td>1454</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>AWS managed policies</td>
<td>1457</td>
</tr>
<tr>
<td>Troubleshooting</td>
<td>1462</td>
</tr>
<tr>
<td>Governance</td>
<td>1465</td>
</tr>
<tr>
<td>Proactive controls with Guard</td>
<td>1467</td>
</tr>
<tr>
<td>Proactive controls with AWS Config</td>
<td>1471</td>
</tr>
<tr>
<td>Detective controls with AWS Config</td>
<td>1478</td>
</tr>
<tr>
<td>Code signing</td>
<td>1482</td>
</tr>
<tr>
<td>Code scanning</td>
<td>1485</td>
</tr>
<tr>
<td>Observability</td>
<td>1490</td>
</tr>
<tr>
<td>Compliance validation</td>
<td>1497</td>
</tr>
<tr>
<td>Resilience</td>
<td>1497</td>
</tr>
<tr>
<td>Infrastructure security</td>
<td>1498</td>
</tr>
<tr>
<td><strong>Monitoring functions</strong></td>
<td>1499</td>
</tr>
<tr>
<td>Monitoring console</td>
<td>1500</td>
</tr>
<tr>
<td>Pricing</td>
<td>1500</td>
</tr>
<tr>
<td>Using the Lambda console</td>
<td>1500</td>
</tr>
<tr>
<td>Types of monitoring graphs</td>
<td>1500</td>
</tr>
<tr>
<td>Viewing graphs on the Lambda console</td>
<td>1501</td>
</tr>
<tr>
<td>Viewing queries on the CloudWatch Logs console</td>
<td>1502</td>
</tr>
<tr>
<td>What's next?</td>
<td>1503</td>
</tr>
<tr>
<td>Function metrics</td>
<td>1504</td>
</tr>
<tr>
<td>Viewing metrics on the CloudWatch console</td>
<td>1504</td>
</tr>
<tr>
<td>Types of metrics</td>
<td>1505</td>
</tr>
<tr>
<td>Function logs</td>
<td>1509</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>1510</td>
</tr>
<tr>
<td>Pricing</td>
<td>1510</td>
</tr>
<tr>
<td>Configuring advanced logging controls for your Lambda function</td>
<td>1510</td>
</tr>
<tr>
<td>Using the Lambda console</td>
<td>1524</td>
</tr>
<tr>
<td>Using the AWS CLI</td>
<td>1524</td>
</tr>
<tr>
<td>Runtime function logging</td>
<td>1527</td>
</tr>
<tr>
<td>What's next?</td>
<td>1528</td>
</tr>
<tr>
<td>AWS X-Ray</td>
<td>1529</td>
</tr>
<tr>
<td>Execution role permissions</td>
<td>1532</td>
</tr>
<tr>
<td>The AWS X-Ray daemon</td>
<td>1532</td>
</tr>
<tr>
<td>Enabling active tracing with the Lambda API</td>
<td>1533</td>
</tr>
<tr>
<td>Enabling active tracing with AWS CloudFormation</td>
<td>1533</td>
</tr>
</tbody>
</table>
Lambda: Concurrency and memory quotas ................................................................. 1650
Invocation .................................................................................................................. 1650
IAM: lambda:InvokeFunction not authorized ......................................................... 1651
Lambda: Couldn't find valid bootstrap (Runtime.InvalidEntrypoint) ...................... 1651
Lambda: Operation cannot be performed ResourceConflictException .................... 1651
Lambda: Function is stuck in Pending ................................................................. 1652
Lambda: One function is using all concurrency .................................................... 1652
General: Cannot invoke function with other accounts or services ....................... 1652
General: Function invocation is looping .............................................................. 1652
Lambda: Alias routing with provisioned concurrency .......................................... 1653
Lambda: Cold starts with provisioned concurrency ............................................. 1653
Lambda: Cold starts with new versions ................................................................. 1654
EFS: Function could not mount the EFS file system ............................................ 1654
EFS: Function could not connect to the EFS file system ..................................... 1654
EFS: Function could not mount the EFS file system due to timeout ................... 1655
Lambda: Lambda detected an IO process that was taking too long ....................... 1655
Execution .................................................................................................................. 1655
Lambda: Execution takes too long ........................................................................ 1656
Lambda: Logs or traces don't appear ...................................................................... 1656
Lambda: Not all of my function's logs appear ....................................................... 1656
Lambda: The function returns before execution finishes ..................................... 1657
AWS SDK: Versions and updates ......................................................................... 1657
Python: Libraries load incorrectly ........................................................................ 1658
Networking .............................................................................................................. 1658
VPC: Function loses internet access or times out ............................................... 1659
VPC: Function needs access to AWS services without using the internet .......... 1659
VPC: Elastic network interface limit reached ...................................................... 1659
EC2: Elastic network interface with type of "lambda" .......................................... 1660
Container images ................................................................................................... 1660
Container: CodeArtifactUserException errors related to the code artifact .......... 1660
Container: ManifestKeyCustomerException errors related to the code manifest key 1660
Container: Error occurs on runtime InvalidEntrypoint ...................................... 1661
Lambda: System provisioning additional capacity ............................................... 1661
CloudFormation: ENTRYPOINT is being overridden with a null or empty value .... 1661

**Lambda applications** .............................................................................................. 1663

Manage applications ............................................................................................... 1665
Monitoring applications ................................................................. 1665
Custom monitoring dashboards ..................................................... 1666
Tutorial – Create an application ..................................................... 1668
Prerequisites .............................................................................. 1669
Create an application .................................................................. 1670
Invoke the function .................................................................... 1671
Add an AWS resource .................................................................. 1672
Update the permissions boundary ................................................. 1674
Update the function code ............................................................. 1675
Next steps .................................................................................. 1677
Troubleshooting ......................................................................... 1678
Clean up ...................................................................................... 1679
Rolling deployments .................................................................... 1681
Example AWS SAM Lambda template ............................................ 1681
Mobile SDK for Android ................................................................ 1683
Tutorial ....................................................................................... 1683
Sample code ............................................................................... 1692
Orchestrating functions ............................................................... 1695
Application patterns .................................................................... 1695
State machine components .......................................................... 1695
State machine application patterns ............................................... 1696
Applying patterns to state machines .............................................. 1696
Example branching application pattern ......................................... 1697
Manage state machines ............................................................... 1700
Viewing state machine details ...................................................... 1700
Editing a state machine ............................................................... 1701
Running a state machine ............................................................. 1702
Orchestration examples ............................................................... 1702
Configuring a Lambda function as a task ...................................... 1702
Configuring a state machine as an event source ......................... 1703
Handling function and service errors ............................................ 1704
AWS CloudFormation and AWS SAM ......................................... 1705
Lambda SnapStart ........................................................................ 1708
Supported features and limitations ............................................... 1708
Supported Regions ...................................................................... 1709
Compatibility considerations ...................................................... 1710
Instrumentation with AWS X-Ray .................................................. 1743
AWS CloudFormation template and additional resources ........................................... 1743
List manager .................................................................................. 1745
Architecture and event structure .............................................................................. 1745
Instrumentation with AWS X-Ray ........................................................................... 1748
AWS CloudFormation templates and additional resources ...................................... 1748
Working with AWS SDKs ............................................................................. 1749
Code examples ....................................................................................... 1750
Actions ............................................................................................... 1759
Create a function .................................................................................... 1759
Delete a function ..................................................................................... 1777
Get a function .......................................................................................... 1788
Invoke a function ..................................................................................... 1797
List functions ............................................................................................. 1810
Update function code .................................................................................... 1820
Update function configuration .............................................................................. 1832
Scenarios ............................................................................................... 1842
Get started with functions ............................................................................... 1842
Serverless examples .................................................................................. 1955
Invoke a Lambda function from a Kinesis trigger .................................................. 1955
Invoke a Lambda function from an Amazon S3 trigger .......................................... 1962
Invoke a Lambda function from an Amazon SNS trigger ........................................ 1971
Invoke a Lambda function from an Amazon SQS trigger ...................................... 1975
Reporting batch item failures for Lambda functions with a Kinesis trigger .............. 1982
Reporting batch item failures for Lambda functions with an Amazon SQS trigger ...... 1988
Cross-service examples ............................................................................. 1995
Create a REST API to track COVID-19 data ....................................................... 1995
Create a lending library REST API ...................................................................... 1996
Create a messenger application ............................................................................ 1997
Create a serverless application to manage photos .................................................. 1998
Create a websocket chat application ..................................................................... 2002
Create an application to analyze customer feedback .............................................. 2003
Invoke a Lambda function from a browser ......................................................... 2009
Use API Gateway to invoke a Lambda function ..................................................... 2010
Use Step Functions to invoke Lambda functions ................................................... 2012
Use scheduled events to invoke a Lambda function .............................................. 2013
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetLayerVersionArn</td>
<td>2181</td>
</tr>
<tr>
<td>GetLayerVersionPolicy</td>
<td>2186</td>
</tr>
<tr>
<td>GetPolicy</td>
<td>2189</td>
</tr>
<tr>
<td>GetProvisionedConcurrencyConfig</td>
<td>2192</td>
</tr>
<tr>
<td>GetRuntimeManagementConfig</td>
<td>2196</td>
</tr>
<tr>
<td>Invoke</td>
<td>2200</td>
</tr>
<tr>
<td>InvokeAsync</td>
<td>2209</td>
</tr>
<tr>
<td>InvokeWithResponseStream</td>
<td>2212</td>
</tr>
<tr>
<td>ListAliases</td>
<td>2220</td>
</tr>
<tr>
<td>ListCodeSigningConfigs</td>
<td>2224</td>
</tr>
<tr>
<td>ListEventSourceMappings</td>
<td>2227</td>
</tr>
<tr>
<td>ListFunctionEventInvokeConfigs</td>
<td>2232</td>
</tr>
<tr>
<td>ListFunctions</td>
<td>2236</td>
</tr>
<tr>
<td>ListFunctionsByCodeSigningConfig</td>
<td>2241</td>
</tr>
<tr>
<td>ListFunctionUrlConfigs</td>
<td>2244</td>
</tr>
<tr>
<td>ListLayers</td>
<td>2248</td>
</tr>
<tr>
<td>ListLayerVersions</td>
<td>2252</td>
</tr>
<tr>
<td>ListProvisionedConcurrencyConfigs</td>
<td>2256</td>
</tr>
<tr>
<td>ListTags</td>
<td>2260</td>
</tr>
<tr>
<td>ListVersionsByFunction</td>
<td>2263</td>
</tr>
<tr>
<td>PublishLayerVersion</td>
<td>2269</td>
</tr>
<tr>
<td>PublishVersion</td>
<td>2276</td>
</tr>
<tr>
<td>PutFunctionCodeSigningConfig</td>
<td>2289</td>
</tr>
<tr>
<td>PutFunctionConcurrency</td>
<td>2293</td>
</tr>
<tr>
<td>PutFunctionEventInvokeConfig</td>
<td>2297</td>
</tr>
<tr>
<td>PutProvisionedConcurrencyConfig</td>
<td>2303</td>
</tr>
<tr>
<td>PutRuntimeManagementConfig</td>
<td>2308</td>
</tr>
<tr>
<td>RemoveLayerVersionPermission</td>
<td>2313</td>
</tr>
<tr>
<td>RemovePermission</td>
<td>2316</td>
</tr>
<tr>
<td>TagResource</td>
<td>2319</td>
</tr>
<tr>
<td>UntagResource</td>
<td>2322</td>
</tr>
<tr>
<td>UpdateAlias</td>
<td>2325</td>
</tr>
<tr>
<td>UpdateCodeSigningConfig</td>
<td>2331</td>
</tr>
<tr>
<td>UpdateEventSourceMapping</td>
<td>2335</td>
</tr>
<tr>
<td>UpdateFunctionCode</td>
<td>2351</td>
</tr>
<tr>
<td>UpdateFunctionConfiguration</td>
<td>2366</td>
</tr>
</tbody>
</table>
UpdateFunctionEventInvokeConfig ........................................................................................................ 2384
UpdateFunctionUrlConfig ......................................................................................................................... 2389
Data Types .................................................................................................................................................. 2394
AccountLimit ............................................................................................................................................... 2397
AccountUsage ............................................................................................................................................. 2399
AliasConfiguration ..................................................................................................................................... 2400
AliasRoutingConfiguration ......................................................................................................................... 2402
AllowedPublishers ...................................................................................................................................... 2403
AmazonManagedKafkaEventSourceConfig ...................................................................................................... 2404
CodeSigningConfig ..................................................................................................................................... 2405
CodeSigningPolicies .................................................................................................................................... 2407
Concurrency .................................................................................................................................................. 2408
Cors .............................................................................................................................................................. 2409
DeadLetterConfig ....................................................................................................................................... 2412
DestinationConfig ....................................................................................................................................... 2413
DocumentDBEventSourceConfig .................................................................................................................... 2414
Environment .................................................................................................................................................. 2416
EnvironmentError ....................................................................................................................................... 2417
EnvironmentResponse ................................................................................................................................. 2418
EphemeralStorage ....................................................................................................................................... 2419
EventSourceMappingConfiguration ................................................................................................................ 2420
FileSystemConfig ....................................................................................................................................... 2427
Filter ............................................................................................................................................................. 2429
FilterCriteria ............................................................................................................................................... 2430
FunctionCode .............................................................................................................................................. 2431
FunctionCodeLocation ................................................................................................................................. 2433
FunctionConfiguration ................................................................................................................................. 2435
FunctionEventInvokeConfig .......................................................................................................................... 2444
FunctionUrlConfig ...................................................................................................................................... 2446
ImageConfig .................................................................................................................................................. 2449
ImageConfigError ....................................................................................................................................... 2451
ImageConfigResponse ................................................................................................................................. 2452
InvokeResponseStreamUpdate ...................................................................................................................... 2453
InvokeWithResponseStreamCompleteEvent .................................................................................................. 2454
InvokeWithResponseStreamResponseEvent .................................................................................................. 2455
Layer ............................................................................................................................................................ 2456
LayersListItem .................................................................................................................. 2458
LayerVersionContentInput ................................................................. 2460
LayerVersionContentOutput ................................................................. 2462
LayerVersionsListItem ................................................................................................. 2464
LoggingConfig .............................................................................................................. 2467
OnFailure ...................................................................................................................... 2469
OnSuccess ..................................................................................................................... 2470
ProvisionedConcurrentConfigListItem ................................................................. 2471
RuntimeVersionConfig ................................................................................................. 2473
RuntimeVersionError ................................................................................................. 2474
ScalingConfig .............................................................................................................. 2475
SelfManagedEventSource ............................................................................................ 2476
SelfManagedKafkaEventSourceConfig ................................................................. 2477
SnapStart ..................................................................................................................... 2478
SnapStartResponse ....................................................................................................... 2479
SourceAccessConfiguration ......................................................................................... 2480
TracingConfig .............................................................................................................. 2482
TracingConfigResponse ............................................................................................... 2483
VpcConfig .................................................................................................................... 2484
VpcConfigResponse ...................................................................................................... 2486
Certificate errors when using an SDK ........................................................................ 2487
Common Errors ........................................................................................................... 2488
Common Parameters ................................................................................................. 2489
Document history ......................................................................................................... 2492
Earlier updates ............................................................................................................ 2514
What is AWS Lambda?

AWS Lambda is a compute service that lets you run code without provisioning or managing servers.

Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, and logging. With Lambda, all you need to do is supply your code in one of the language runtimes that Lambda supports.

You organize your code into Lambda functions. The Lambda service runs your function only when needed and scales automatically. You only pay for the compute time that you consume—there is no charge when your code is not running. For more information, see AWS Lambda Pricing.

Tip

To learn how to build serverless solutions, check out the Serverless Developer Guide.

When to use Lambda

Lambda is an ideal compute service for application scenarios that need to scale up rapidly, and scale down to zero when not in demand. For example, you can use Lambda for:

- **File processing**: Use Amazon Simple Storage Service (Amazon S3) to trigger Lambda data processing in real time after an upload.

- **Stream processing**: Use Lambda and Amazon Kinesis to process real-time streaming data for application activity tracking, transaction order processing, clickstream analysis, data cleansing, log filtering, indexing, social media analysis, Internet of Things (IoT) device data telemetry, and metering.

- **Web applications**: Combine Lambda with other AWS services to build powerful web applications that automatically scale up and down and run in a highly available configuration across multiple data centers.

- **IoT backends**: Build serverless backends using Lambda to handle web, mobile, IoT, and third-party API requests.

- **Mobile backends**: Build backends using Lambda and Amazon API Gateway to authenticate and process API requests. Use AWS Amplify to easily integrate with your iOS, Android, Web, and React Native frontends.
When using Lambda, you are responsible only for your code. Lambda manages the compute fleet that offers a balance of memory, CPU, network, and other resources to run your code. Because Lambda manages these resources, you cannot log in to compute instances or customize the operating system on provided runtimes.

Lambda performs operational and administrative activities on your behalf, including managing capacity, monitoring, and logging your Lambda functions.

If you do need to manage your compute resources, AWS has other compute services to consider, such as:

- AWS App Runner builds and deploys containerized web applications automatically, load balances traffic with encryption, scales to meet your traffic needs, and allows for the configuration of how services are accessed and communicate with other AWS applications in a private Amazon VPC.
- AWS Fargate with Amazon ECS runs containers without having to provision, configure, or scale clusters of virtual machines.
- Amazon EC2 lets you customize operating system, network and security settings, and the entire software stack. You are responsible for provisioning capacity, monitoring fleet health and performance, and using Availability Zones for fault tolerance.

**Key features**

The following key features help you develop Lambda applications that are scalable, secure, and easily extensible:

**Configuring function options**

Configure your Lambda function using the console or AWS CLI.

**Environment variables**

Use environment variables to adjust your function’s behavior without updating code.

**Versions**

Manage the deployment of your functions with versions, so that, for example, a new function can be used for beta testing without affecting users of the stable production version.
Container images

Create a container image for a Lambda function by using an AWS provided base image or an alternative base image so that you can reuse your existing container tooling or deploy larger workloads that rely on sizable dependencies, such as machine learning.

Layers

Package libraries and other dependencies to reduce the size of deployment archives and makes it faster to deploy your code.

Lambda extensions

Augment your Lambda functions with tools for monitoring, observability, security, and governance.

Function URLs

Add a dedicated HTTP(S) endpoint to your Lambda function.

Response streaming

Configure your Lambda function URLs to stream response payloads back to clients from Node.js functions, to improve time to first byte (TTFB) performance or to return larger payloads.

Concurrency and scaling controls

Apply fine-grained control over the scaling and responsiveness of your production applications.

Code signing

Verify that only approved developers publish unaltered, trusted code in your Lambda functions.

Private networking

Create a private network for resources such as databases, cache instances, or internal services.

File system access

Configure a function to mount an Amazon Elastic File System (Amazon EFS) to a local directory, so that your function code can access and modify shared resources safely and at high concurrency.

Lambda SnapStart for Java

Improve startup performance for Java runtimes by up to 10x at no extra cost, typically with no changes to your function code.
Getting started with Lambda

To get started with Lambda, use the Lambda console to create a function. In a few minutes, you can create and deploy a function and test it in the console.

As you carry out the tutorial, you'll learn some fundamental Lambda concepts, like how to pass arguments to your function using the Lambda event object. You'll also learn how to return log outputs from your function, and how to view your function's invocation logs in CloudWatch Logs.

To keep things simple, you create your function using either the Python or Node.js runtime. With these interpreted languages, you can edit function code directly in the console's built-in code editor. With compiled languages like Java and C#, you need to create a deployment package on your local build machine and upload it to Lambda. To learn about deploying functions to Lambda using other runtimes, see the links in the section called “Additional resources and next steps” section.

Tip

To learn how to build serverless solutions, check out the Serverless Developer Guide.

Prerequisites

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an AWS account root user is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to an administrative user, and use only the root user to perform tasks that require root user access.
AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing My Account.

Create an administrative user

After you sign up for an AWS account, create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing Root user and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the AWS Sign-In User Guide.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the IAM User Guide.

Create an administrative user

• For your daily administrative tasks, grant administrative access to an administrative user in AWS IAM Identity Center.

   For instructions, see Getting started in the AWS IAM Identity Center User Guide.

Sign in as the administrative user

• To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

   For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the AWS Sign-In User Guide.
Create a Lambda function with the console

In this example, your function takes a JSON object containing two integer values labeled "length" and "width". The function multiplies these values to calculate an area and returns this as a JSON string.

Your function also prints the calculated area, along with the name of its CloudWatch log group. Later in the tutorial, you’ll learn to use CloudWatch Logs to view records of your functions’ invocation.

To create your function, you first use the console to create a basic Hello world function. In the following step, you then add your own function code.

To create a Hello world Lambda function with the console

1. Open the Functions page of the Lambda console.
2. Choose Create function.
3. Select Author from scratch.
4. In the Basic information pane, for Function name enter myLambdaFunction.
5. For Runtime, choose either Node.js 18.x or Python 3.11
6. Leave architecture set to x86_64 and choose Create function.

Lambda creates a function that returns the message Hello from Lambda! Lambda also creates an execution role for your function. An execution role is an AWS Identity and Access Management (IAM) role that grants a Lambda function permission to access AWS services and resources. For your function, the role that Lambda creates grants basic permissions to write to CloudWatch Logs.

You now use the console’s built-in code editor to replace the Hello world code that Lambda created with your own function code.

Node.js

To modify the code in the console

1. Choose the Code tab.
In the console's built-in code editor, you should see the function code that Lambda created. If you don't see the `index.mjs` tab in the code editor, select `index.mjs` in the file explorer as shown on the following diagram.

2. Paste the following code into the `index.mjs` tab, replacing the code that Lambda created.

```javascript
export const handler = async (event, context) => {

    const length = event.length;
    const width = event.width;
    var area = calculateArea(length, width);
    console.log(`The area is ${area}`);

    console.log('CloudWatch log group: ', context.logGroupName);

    const data = {
        "area": area,
    }
    return JSON.stringify(data);

    function calculateArea(length, width) {
        return length * width;
    }
};
```

3. Select Deploy to update your function's code. When Lambda has deployed the changes, the console displays a banner letting you know that it's successfully updated your function.
Understanding your function code

Before you move to the next step, let’s take a moment to look at the function code and understand some key Lambda concepts.

- The Lambda handler:

Your Lambda function contains a Node.js function named `handler`. A Lambda function in Node.js can contain more than one Node.js function, but the `handler` function is always the entry point to your code. When your function is invoked, Lambda runs this method.

When you created your Hello world function using the console, Lambda automatically set the name of the handler method for your function to `handler`. Be sure not to edit the name of this Node.js function. If you do, Lambda won’t be able to run your code when you invoke your function.

To learn more about the Lambda handler in Node.js, see the section called “Handler”.

- The Lambda event object:

The function `handler` takes two arguments, `event` and `context`. An `event` in Lambda is a JSON formatted document that contains data for your function to process.

If your function is invoked by another AWS service, the event object contains information about the event that caused the invocation. For example, if an Amazon Simple Storage Service (Amazon S3) bucket invokes your function when an object is uploaded, the event will contain the name of the Amazon S3 bucket and the object key.

In this example, you’ll create an event in the console by entering a JSON formatted document with two key-value pairs.

- The Lambda context object:

The second argument your function takes is `context`. Lambda passes the `context object` to your function automatically. The context object contains information about the function invocation and execution environment.

You can use the context object to output information about your function's invocation for monitoring purposes. In this example, your function uses the `logGroupName` parameter to output the name of its CloudWatch log group.

To learn more about the Lambda context object in Node.js, see the section called “Context”.
 Logging in Lambda:

With Node.js, you can use console methods like `console.log` and `console.error` to send information to your function's log. The example code uses `console.log` statements to output the calculated area and the name of the function's CloudWatch Logs group. You can also use any logging library that writes to stdout or stderr.

To learn more, see the section called “Logging”. To learn about logging in other runtimes, see the 'Building with' pages for the runtimes you're interested in.

Python

To modify the code in the console

1. Choose the Code tab.

   In the console's built-in code editor, you should see the function code that Lambda created. If you don't see the `lambda_function.py` tab in the code editor, select `lambda_function.py` in the file explorer as shown on the following diagram.

2. Paste the following code into the `lambda_function.py` tab, replacing the code that Lambda created.

   ```python
   import json
   import logging
   ```
logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    # Get the length and width parameters from the event object. The
    # runtime converts the event object to a Python dictionary
    length=event['length']
    width=event['width']
    area = calculate_area(length, width)
    print(f"The area is {area}")
    logger.info(f"CloudWatch logs group: {context.log_group_name}")

    # return the calculated area as a JSON string
    data = {"area": area}
    return json.dumps(data)

def calculate_area(length, width):
    return length*width

3. Select **Deploy** to update your function's code. When Lambda has deployed the changes, the
   console displays a banner letting you know that it's successfully updated your function.

**Understanding your function code**

Before you move to the next step, let's take a moment to look at the function code and understand some key Lambda concepts.

- The Lambda handler:

  Your Lambda function contains a Python function named `lambda_handler`. A Lambda function in Python can contain more than one Python function, but the *handler* function is always the entry point to your code. When your function is invoked, Lambda runs this method.

  When you created your Hello world function using the console, Lambda automatically set the name of the handler method for your function to `lambda_handler`. Be sure not to edit the name of this Python function. If you do, Lambda won’t be able to run your code when you invoke your function.
To learn more about the Lambda handler in Python, see the section called “Handler”.

- The Lambda event object:

  The function `lambda_handler` takes two arguments, `event` and `context`. An `event` in Lambda is a JSON formatted document that contains data for your function to process.

  If your function is invoked by another AWS service, the event object contains information about the event that caused the invocation. For example, if an Amazon Simple Storage Service (Amazon S3) bucket invokes your function when an object is uploaded, the event will contain the name of the Amazon S3 bucket and the object key.

  In this example, you’ll create an event in the console by entering a JSON formatted document with two key-value pairs.

- The Lambda context object:

  The second argument your function takes is `context`. Lambda passes the `context object` to your function automatically. The context object contains information about the function invocation and execution environment.

  You can use the context object to output information about your function's invocation for monitoring purposes. In this example, your function uses the `log_group_name` parameter to output the name of its CloudWatch log group.

  To learn more about the Lambda context object in Python, see the section called “Context”.

- Logging in Lambda:

  With Python, you can use either a `print` statement or a Python logging library to send information to your function's log. To illustrate the difference in what's captured, the example code uses both methods. In a production application, we recommend that you use a logging library.

  To learn more, see the section called “Logging”. To learn about logging in other runtimes, see the 'Building with' pages for the runtimes you're interested in.
Invoke the Lambda function using the console

To invoke your function using the Lambda console, you first create a test event to send to your function. The event is a JSON formatted document containing two key-value pairs with the keys "length" and "width".

**To create the test event**

1. In the **Code source** pane, choose **Test**.
2. Select **Create new event**.
3. For **Event name** enter **myTestEvent**.
4. In the **Event JSON** panel, replace the default values by pasting in the following:

   ```json
   {
       "length": 6,
       "width": 7
   }
   
   5. Choose **Save**.

You now test your function and use the Lambda console and CloudWatch Logs to view records of your function's invocation.

**To test your function and view invocation records in the console**

- In the **Code source** pane, choose **Test**. When your function finishes running, you’ll see the response and function logs displayed in the **Execution results** tab. You should see results similar to the following.

   **Node.js**

<table>
<thead>
<tr>
<th>Test Event Name</th>
<th>myTestEvent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response</td>
<td>&quot;{{&quot;area&quot;:42}&quot;</td>
</tr>
<tr>
<td>Function Logs</td>
<td>START RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Version: $LATEST</td>
</tr>
</tbody>
</table>
In this example, you invoked your code using the console's test feature. This means that you can view your function's execution results directly in the console. When your function is invoked outside the console, you need to use CloudWatch Logs.

**To view your function's invocation records in CloudWatch Logs**

1. Open the [Log groups](#) page of the CloudWatch console.
2. Choose the log group for your function (/aws/lambda/myLambdaFunction). This is the log group name that your function printed to the console.

3. In the **Log streams** tab, choose the log stream for your function's invocation.

You should see output similar to the following:

**Node.js**

```
INIT_START Runtime Version: nodejs:18.v10    Runtime Version ARN: arn:aws:lambda:us-west-2::runtime:e3aaabf6b92ef8755eaae2f4bdfcb7eb8c4536a5e044900570a42bdab7b869d9
START RequestId: aba6c0fc-cf99-49d7-a77d-26d805dadc20 Version: $LATEST
2023-08-23T22:04:15.809Z    5c012b0a-18f7-4805-b2f6-40912935034a  INFO The area is 42
2023-08-23T22:04:15.810Z    aba6c0fc-cf99-49d7-a77d-26d805dadc20 INFO CloudWatch log group: /aws/lambda/myLambdaFunction
END RequestId: aba6c0fc-cf99-49d7-a77d-26d805dadc20
REPORT RequestId: aba6c0fc-cf99-49d7-a77d-26d805dadc20    Duration: 17.77 ms
Billed Duration: 18 ms    Memory Size: 128 MB    Max Memory Used: 67 MB    Init Duration: 178.85 ms
```

**Python**

```
INIT_START Runtime Version: python:3.11.v9    Runtime Version ARN: arn:aws:lambda:us-west-2::runtime:ca202755c87b9ec2b58856efb7374b4f7b655a0ea3de8b1d5acc9ae9e297b072
START RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e Version: $LATEST
The area is 42
[INFO] 2023-09-01T00:05:22.464Z 9315ab6b-354a-486e-884a-2fb2972b7d84 CloudWatch logs group: /aws/lambda/myLambdaFunction
END RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e
REPORT RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e    Duration: 1.15 ms
Billed Duration: 2 ms    Memory Size: 128 MB    Max Memory Used: 40 MB
```

**Clean up**

When you're finished working with the example function, delete it. You can also delete the log group that stores the function's logs, and the **execution role** that the console created.
**To delete a Lambda function**

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Actions, Delete**.
4. In the **Delete function** dialog box, enter `delete`, and then choose **Delete**.

**To delete the log group**

1. Open the [Log groups page](#) of the CloudWatch console.
2. Select the function's log group (`/aws/lambda/my-function`).
3. Choose **Actions, Delete log group(s)**.
4. In the **Delete log group(s)** dialog box, choose **Delete**.

**To delete the execution role**

1. Open the [Roles page](#) of the AWS Identity and Access Management (IAM) console.
2. Select the function's execution role (for example, `myLambdaFunction-role-31exmpl`).
3. Choose **Delete**.
4. In the **Delete role** dialog box, enter the role name and then choose **Delete**.

You can automate the creation and cleanup of functions, log groups, and roles with AWS CloudFormation and the AWS Command Line Interface (AWS CLI).

**Additional resources and next steps**

Now you’ve created and tested a simple Lambda function using the console, take these next steps:

- Learn to add dependencies to your code and deploy it using a .zip deployment package. Choose from the following links for the languages you're interested in.
  - **Node.js**

    See [the section called “Deploy .zip file archives”](#)
Typescript

See the section called “Deploy .zip file archives”

Python

See the section called “Deploy .zip file archives”

Ruby

See the section called “Deploy .zip file archives”

Java

See the section called “Deploy .zip file archives”

Go

See the section called “Deploy .zip file archives”

C#

See the section called “Deployment package”

- Carry out the tutorial Using an Amazon S3 trigger to invoke a Lambda function to learn how to configure a Lambda function to be invoked by another AWS service.
- Choose one of the following tutorials for a more complex example of using Lambda with other AWS services.
  - Using Lambda with API Gateway: Create an Amazon API Gateway REST API that invokes a Lambda function.
  - Using a Lambda function to access an Amazon RDS database: Use a Lambda function to write data to an Amazon Relational Database Service (Amazon RDS) database through RDS Proxy.
  - Using an Amazon S3 trigger to create thumbnail images: Use a Lambda function to create a thumbnail every time an image file is uploaded to an Amazon S3 bucket.
AWS Lambda foundations

The Lambda function is the principal resource of the Lambda service.

You can configure your functions using the Lambda console, Lambda API, AWS CloudFormation or AWS SAM. You create code for the function and upload the code using a deployment package. Lambda invokes the function when an event occurs. Lambda runs multiple instances of your function in parallel, governed by concurrency and scaling limits.

Topics

- **Lambda concepts**
- **Lambda programming model**
- **Lambda execution environment**
- **Lambda deployment packages**
- **Using Lambda with infrastructure as code (IaC)**
- **Private networking with VPC**
- **Lambda console**
- **Lambda instruction set architectures (ARM/x86)**
- **Additional Lambda features**
- **Learn how to build serverless solutions**
Lambda concepts

Lambda runs instances of your function to process events. You can invoke your function directly using the Lambda API, or you can configure an AWS service or resource to invoke your function.

Concepts

- **Function**
- **Trigger**
- **Event**
- **Execution environment**
- **Instruction set architecture**
- **Deployment package**
- **Runtime**
- **Layer**
- **Extension**
- **Concurrency**
- **Qualifier**
- **Destination**

**Function**

A *function* is a resource that you can invoke to run your code in Lambda. A function has code to process the *events* that you pass into the function or that other AWS services send to the function.

**Trigger**

A *trigger* is a resource or configuration that invokes a Lambda function. Triggers include AWS services that you can configure to invoke a function and *event source mappings*. An event source mapping is a resource in Lambda that reads items from a stream or queue and invokes a function. For more information, see [Invoking Lambda functions](#) and [Using AWS Lambda with other services](#).
Event

An event is a JSON-formatted document that contains data for a Lambda function to process. The runtime converts the event to an object and passes it to your function code. When you invoke a function, you determine the structure and contents of the event.

Example custom event – weather data

```json
{
  "TemperatureK": 281,
  "WindKmh": -3,
  "HumidityPct": 0.55,
  "PressureHPa": 1020
}
```

When an AWS service invokes your function, the service defines the shape of the event.

Example service event – Amazon SNS notification

```json
{
  "Records": [
    {
      "Sns": {
        "Timestamp": "2019-01-02T12:45:07.000Z",
        "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi+tE/1+82j...65r==",
        "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
        "Message": "Hello from SNS!",
        ...
      }
    }
  ]
}
```

For more information about events from AWS services, see [Using AWS Lambda with other services](#).

Execution environment

An execution environment provides a secure and isolated runtime environment for your Lambda function. An execution environment manages the processes and resources that are required to run the function. The execution environment provides lifecycle support for the function and for any extensions associated with your function.

For more information, see [Lambda execution environment](#).
Instruction set architecture

The instruction set architecture determines the type of computer processor that Lambda uses to run the function. Lambda provides a choice of instruction set architectures:

- arm64 – 64-bit ARM architecture, for the AWS Graviton2 processor.
- x86_64 – 64-bit x86 architecture, for x86-based processors.

For more information, see Lambda instruction set architectures (ARM/x86).

Deployment package

You deploy your Lambda function code using a deployment package. Lambda supports two types of deployment packages:

- A .zip file archive that contains your function code and its dependencies. Lambda provides the operating system and runtime for your function.
- A container image that is compatible with the Open Container Initiative (OCI) specification. You add your function code and dependencies to the image. You must also include the operating system and a Lambda runtime.

For more information, see Lambda deployment packages.

Runtime

The runtime provides a language-specific environment that runs in an execution environment. The runtime relays invocation events, context information, and responses between Lambda and the function. You can use runtimes that Lambda provides, or build your own. If you package your code as a .zip file archive, you must configure your function to use a runtime that matches your programming language. For a container image, you include the runtime when you build the image.

For more information, see Lambda runtimes.

Layer

A Lambda layer is a .zip file archive that can contain additional code or other content. A layer can contain libraries, a custom runtime, data, or configuration files.
Layers provide a convenient way to package libraries and other dependencies that you can use with your Lambda functions. Using layers reduces the size of uploaded deployment archives and makes it faster to deploy your code. Layers also promote code sharing and separation of responsibilities so that you can iterate faster on writing business logic.

You can include up to five layers per function. Layers count towards the standard Lambda deployment size quotas. When you include a layer in a function, the contents are extracted to the /opt directory in the execution environment.

By default, the layers that you create are private to your AWS account. You can choose to share a layer with other accounts or to make the layer public. If your functions consume a layer that a different account published, your functions can continue to use the layer version after it has been deleted, or after your permission to access the layer is revoked. However, you cannot create a new function or update functions using a deleted layer version.

Functions deployed as a container image do not use layers. Instead, you package your preferred runtime, libraries, and other dependencies into the container image when you build the image.

For more information, see [Lambda layers](https://aws.amazon.com/blogs/compute/lambda-layers/).

**Extension**

Lambda extensions enable you to augment your functions. For example, you can use extensions to integrate your functions with your preferred monitoring, observability, security, and governance tools. You can choose from a broad set of tools that [AWS Lambda Partners](https://aws.amazon.com/partners/lambda/) provides, or you can [create your own Lambda extensions](https://aws.amazon.com/blogs/compute/lambda-extensions/).

An internal extension runs in the runtime process and shares the same lifecycle as the runtime. An external extension runs as a separate process in the execution environment. The external extension is initialized before the function is invoked, runs in parallel with the function's runtime, and continues to run after the function invocation is complete.

For more information, see [Lambda extensions](https://aws.amazon.com/blogs/compute/lambda-extensions/).

**Concurrency**

*Concurrency* is the number of requests that your function is serving at any given time. When your function is invoked, Lambda provisions an instance of it to process the event. When the function code finishes running, it can handle another request. If the function is invoked again
while a request is still being processed, another instance is provisioned, increasing the function's concurrency.

Concurrency is subject to quotas at the AWS Region level. You can configure individual functions to limit their concurrency, or to enable them to reach a specific level of concurrency. For more information, see Configuring reserved concurrency.

**Qualifier**

When you invoke or view a function, you can include a qualifier to specify a version or alias. A version is an immutable snapshot of a function's code and configuration that has a numerical qualifier. For example, my-function:1. An alias is a pointer to a version that you can update to map to a different version, or split traffic between two versions. For example, my-function:BLUE. You can use versions and aliases together to provide a stable interface for clients to invoke your function.

For more information, see Lambda function versions.

**Destination**

A destination is an AWS resource where Lambda can send events from an asynchronous invocation. You can configure a destination for events that fail processing. Some services also support a destination for events that are successfully processed.

For more information, see Configuring destinations for asynchronous invocation.
Lambda programming model

Lambda provides a programming model that is common to all of the runtimes. The programming model defines the interface between your code and the Lambda system. You tell Lambda the entry point to your function by defining a *handler* in the function configuration. The runtime passes in objects to the handler that contain the invocation *event* and the *context*, such as the function name and request ID.

When the handler finishes processing the first event, the runtime sends it another. The function's class stays in memory, so clients and variables that are declared outside of the handler method in *initialization code* can be reused. To save processing time on subsequent events, create reusable resources like AWS SDK clients during initialization. Once initialized, each instance of your function can process thousands of requests.

Your function also has access to local storage in the `/tmp` directory. The directory content remains when the execution environment is frozen, providing a transient cache that can be used for multiple invocations. For more information, see Lambda execution environment.

When AWS X-Ray tracing is enabled, the runtime records separate subsegments for initialization and execution.

The runtime captures logging output from your function and sends it to Amazon CloudWatch Logs. In addition to logging your function's output, the runtime also logs entries when function invocation starts and ends. This includes a report log with the request ID, billed duration, initialization duration, and other details. If your function throws an error, the runtime returns that error to the invoker.

⚠️ **Note**

Logging is subject to CloudWatch Logs quotas. Log data can be lost due to throttling or, in some cases, when an instance of your function is stopped.

Lambda scales your function by running additional instances of it as demand increases, and by stopping instances as demand decreases. This model leads to variations in application architecture, such as:

- Unless noted otherwise, incoming requests might be processed out of order or concurrently.
• Do not rely on instances of your function being long lived, instead store your application's state elsewhere.

• Use local storage and class-level objects to increase performance, but keep to a minimum the size of your deployment package and the amount of data that you transfer onto the execution environment.

For a hands-on introduction to the programming model in your preferred programming language, see the following chapters.

• Building Lambda functions with Node.js
• Building Lambda functions with Python
• Building Lambda functions with Ruby
• Building Lambda functions with Java
• Building Lambda functions with Go
• Building Lambda functions with C#
• Building Lambda functions with PowerShell
Lambda execution environment

Lambda invokes your function in an execution environment, which provides a secure and isolated runtime environment. The execution environment manages the resources required to run your function. The execution environment also provides lifecycle support for the function's runtime and any external extensions associated with your function.

The function's runtime communicates with Lambda using the Runtime API. Extensions communicate with Lambda using the Extensions API. Extensions can also receive log messages and other telemetry from the function by using the Telemetry API.

When you create your Lambda function, you specify configuration information, such as the amount of memory available and the maximum execution time allowed for your function. Lambda uses this information to set up the execution environment.

The function's runtime and each external extension are processes that run within the execution environment. Permissions, resources, credentials, and environment variables are shared between the function and the extensions.

Topics

- Lambda execution environment lifecycle
Lambda execution environment lifecycle

Each phase starts with an event that Lambda sends to the runtime and to all registered extensions. The runtime and each extension indicate completion by sending a Next API request. Lambda freezes the execution environment when the runtime and each extension have completed and there are no pending events.

Topics
- Init phase
- Failures during the Init phase
- Restore phase (Lambda SnapStart only)
- Invoke phase
- Failures during the invoke phase
- Shutdown phase

Init phase

In the Init phase, Lambda performs three tasks:

- Start all extensions (Extension init)
- Bootstrap the runtime (Runtime init)
- Run the function's static code (Function init)
- Run any beforeCheckpoint runtime hooks (Lambda SnapStart only)

The Init phase ends when the runtime and all extensions signal that they are ready by sending a Next API request. The Init phase is limited to 10 seconds. If all three tasks do not complete within 10 seconds, Lambda retries the Init phase at the time of the first function invocation with the configured function timeout.
When **Lambda SnapStart** is activated, the **Init** phase happens when you publish a function version. Lambda saves a snapshot of the memory and disk state of the initialized execution environment, persists the encrypted snapshot, and caches it for low-latency access. If you have a `beforeCheckpoint` runtime hook, then the code runs at the end of **Init** phase.

Note

The 10-second timeout doesn't apply to SnapStart functions. When Lambda creates a snapshot, your initialization code can run for up to 15 minutes. The time limit is 130 seconds or the **configured function timeout** (maximum 900 seconds), whichever is higher.

When you use **provisioned concurrency**, Lambda initializes the execution environment when you configure the PC settings for a function. Lambda also ensures that initialized execution environments are always available in advance of invocations. You may see gaps between your function's invocation and initialization phases. Depending on your function's runtime and memory configuration, you may also see variable latency on the first invocation on an initialized execution environment.

For functions using on-demand concurrency, Lambda may occasionally initialize execution environments ahead of invocation requests. When this happens, you may also observe a time gap between your function's initialization and invocation phases. We recommend you to not take a dependency on this behavior.

### Failures during the **Init** phase

If a function crashes or times out during the **Init** phase, Lambda emits error information in the **INIT_REPORT** log.

**Example — INIT_REPORT log for timeout**

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: timeout
```

**Example — INIT_REPORT log for extension failure**

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: error Error Type: Extension.Crash
```
If the Init phase is successful, Lambda doesn't emit the INIT_REPORT log—unless SnapStart is activated. SnapStart functions always emit INIT_REPORT. For more information, see Monitoring for Lambda SnapStart.

**Restore phase (Lambda SnapStart only)**

When you first invoke a SnapStart function and as the function scales up, Lambda resumes new execution environments from the persisted snapshot instead of initializing the function from scratch. If you have an afterRestore() runtime hook, the code runs at the end of the Restore phase. You are charged for the duration of afterRestore() runtime hooks. The runtime (JVM) must load and afterRestore() runtime hooks must complete within the timeout limit (10 seconds). Otherwise, you'll get a SnapStartTimeoutException. When the Restore phase completes, Lambda invokes the function handler (the **Invoke phase**).

**Failures during the Restore phase**

If the Restore phase fails, Lambda emits error information in the RESTORE_REPORT log.

**Example — RESTORE_REPORT log for timeout**

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: timeout
```

**Example — RESTORE_REPORT log for runtime hook failure**

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: error Error Type: Runtime.ExitError
```

For more information about the RESTORE_REPORT log, see Monitoring for Lambda SnapStart.

**Invoke phase**

When a Lambda function is invoked in response to a Next API request, Lambda sends an Invoke event to the runtime and to each extension.

The function's timeout setting limits the duration of the entire Invoke phase. For example, if you set the function timeout as 360 seconds, the function and all extensions need to complete within 360 seconds. Note that there is no independent post-invoke phase. The duration is the sum of all invocation time (runtime + extensions) and is not calculated until the function and all extensions have finished executing.
The invoke phase ends after the runtime and all extensions signal that they are done by sending a Next API request.

**Failures during the invoke phase**

If the Lambda function crashes or times out during the Invoke phase, Lambda resets the execution environment. The following diagram illustrates Lambda execution environment behavior when there's an invoke failure:

In the previous diagram:

- The first phase is the **INIT** phase, which runs without errors.
- The second phase is the **INVOKE** phase, which runs without errors.
- At some point, suppose your function runs into an invoke failure (such as a function timeout or runtime error). The third phase, labeled **INVOKE WITH ERROR**, illustrates this scenario. When this happens, the Lambda service performs a reset. The reset behaves like a Shutdown event. First, Lambda shuts down the runtime, then sends a Shutdown event to each registered external extension. The event includes the reason for the shutdown. If this environment is used for a new invocation, Lambda re-initializes the extension and runtime together with the next invocation.

  **Note**

  The Lambda reset does not clear the /tmp directory content prior to the next init phase. This behavior is consistent with the regular shutdown phase.

- The fourth phase represents the **INVOKE** phase immediately following an invoke failure. Here, Lambda initializes the environment again by re-running the **INIT** phase. This is called a suppressed init. When suppressed inits occur, Lambda doesn't explicitly report an additional INIT phase in CloudWatch Logs. Instead, you may notice that the duration in the REPORT line includes an additional INIT duration + the INVOKE duration. For example, suppose you see the following logs in CloudWatch:

```
2022-12-20T01:00:00.000-08:00 START RequestId: XXX Version: $LATEST
2022-12-20T01:00:02.500-08:00 END RequestId: XXX
```
In this example, the difference between the REPORT and START timestamps is 2.5 seconds. This doesn't match the reported duration of 3022.91 milliseconds, because it doesn't take into account the extra INIT (suppressed init) that Lambda performed. In this example, you can infer that the actual INVOKE phase took 2.5 seconds.

For more insight into this behavior, you can use the Lambda Telemetry API. The Telemetry API emits INIT_START, INIT_RUNTIME_DONE, and INIT_REPORT events with phase=invoke whenever suppressed inits occur in between invoke phases.

- The fifth phase represents the SHUTDOWN phase, which runs without errors.

**Shutdown phase**

When Lambda is about to shut down the runtime, it sends a Shutdown event to each registered external extension. Extensions can use this time for final cleanup tasks. The Shutdown event is a response to a Next API request.

**Duration:** The entire Shutdown phase is capped at 2 seconds. If the runtime or any extension does not respond, Lambda terminates it via a signal (SIGKILL).

After the function and all extensions have completed, Lambda maintains the execution environment for some time in anticipation of another function invocation. In effect, Lambda freezes the execution environment. When the function is invoked again, Lambda thaws the environment for reuse. Reusing the execution environment has the following implications:

- Objects declared outside of the function's handler method remain initialized, providing additional optimization when the function is invoked again. For example, if your Lambda function establishes a database connection, instead of reestablishing the connection, the original connection is used in subsequent invocations. We recommend adding logic in your code to check if a connection exists before creating a new one.

- Each execution environment provides between 512 MB and 10,240 MB, in 1-MB increments, of disk space in the /tmp directory. The directory content remains when the execution environment is frozen, providing a transient cache that can be used for multiple invocations. You can add extra code to check if the cache has the data that you stored. For more information on deployment size limits, see Lambda quotas.
• Background processes or callbacks that were initiated by your Lambda function and did not complete when the function ended resume if Lambda reuses the execution environment. Make sure that any background processes or callbacks in your code are complete before the code exits.

When you write your function code, do not assume that Lambda automatically reuses the execution environment for subsequent function invocations. Other factors may dictate a need for Lambda to create a new execution environment, which can lead to unexpected results, such as database connection failures.
Lambda deployment packages

Your AWS Lambda function’s code consists of scripts or compiled programs and their dependencies. You use a deployment package to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives.

Topics
- Container images
- .zip file archives
- Layers
- Using other AWS services to build a deployment package

Container images

A container image includes the base operating system, the runtime, Lambda extensions, your application code and its dependencies. You can also add static data, such as machine learning models, into the image.

Lambda provides a set of open-source base images that you can use to build your container image. To create and test container images, you can use the AWS Serverless Application Model (AWS SAM) command line interface (CLI) or native container tools such as the Docker CLI.

You upload your container images to Amazon Elastic Container Registry (Amazon ECR), a managed AWS container image registry service. To deploy the image to your function, you specify the Amazon ECR image URL using the Lambda console, the Lambda API, command line tools, or the AWS SDKs.

For more information about Lambda container images, see Working with Lambda container images.

.zip file archives

A .zip file archive includes your application code and its dependencies. When you author functions using the Lambda console or a toolkit, Lambda automatically creates a .zip file archive of your code.

When you create functions with the Lambda API, command line tools, or the AWS SDKs, you must create a deployment package. You also must create a deployment package if your function uses a
compiled language, or to add dependencies to your function. To deploy your function's code, you upload the deployment package from Amazon Simple Storage Service (Amazon S3) or your local machine.

You can upload a .zip file as your deployment package using the Lambda console, AWS Command Line Interface (AWS CLI), or to an Amazon Simple Storage Service (Amazon S3) bucket.

**Using the Lambda console**

The following steps demonstrate how to upload a .zip file as your deployment package using the Lambda console.

**To upload a .zip file on the Lambda console**

1. Open the [Functions page](#) on the Lambda console.
2. Select a function.
3. In the **Code Source** pane, choose **Upload from** and then **.zip file**.
4. Choose **Upload** to select your local .zip file.
5. Choose **Save**.

**Using the AWS CLI**

You can upload a .zip file as your deployment package using the AWS Command Line Interface (AWS CLI). For language-specific instructions, see the following topics.

**Node.js**

[Deploy Node.js Lambda functions with .zip file archives](#)

**Python**

[Working with .zip file archives for Python Lambda functions](#)

**Ruby**

[Working with .zip file archives for Ruby Lambda functions](#)

**Java**

[Deploy Java Lambda functions with .zip or JAR file archives](#)
Go

**Deploy Go Lambda functions with .zip file archives**

C#

**Build and deploy C# Lambda functions with .zip file archives**

PowerShell

**Deploy PowerShell Lambda functions with .zip file archives**

**Using Amazon S3**

You can upload a .zip file as your deployment package using Amazon Simple Storage Service (Amazon S3). For more information, see [Amazon S3](https://aws.amazon.com/s3/).

**Layers**

If you deploy your function code using a .zip file archive, you can use Lambda layers as a distribution mechanism for libraries, custom runtimes, and other function dependencies. Layers enable you to manage your in-development function code independently from the unchanging code and resources that it uses. You can configure your function to use layers that you create, layers that AWS provides, or layers from other AWS customers.

You do not use layers with container images. Instead, you package your preferred runtime, libraries, and other dependencies into the container image when you build the image.

For more information about layers, see [Lambda layers](https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html).

**Using other AWS services to build a deployment package**

The following section describes other AWS services you can use to package dependencies for your Lambda function.

**Deployment packages with C or C++ libraries**

If your deployment package contains native libraries, you can build the deployment package with AWS Serverless Application Model (AWS SAM). You can use the AWS SAM CLI `sam build` command with the `--use-container` to create your deployment package. This option builds a deployment package inside a Docker image that is compatible with the Lambda execution environment.
For more information, see `sam build` in the *AWS Serverless Application Model Developer Guide*.

**Deployment packages over 50 MB**

If your deployment package is larger than 50 MB, upload your function code and dependencies to an Amazon S3 bucket.

You can create a deployment package and upload the .zip file to your Amazon S3 bucket in the AWS Region where you want to create a Lambda function. When you create your Lambda function, specify the S3 bucket name and object key name on the Lambda console, or using the AWS CLI.

To create a bucket using the Amazon S3 console, see *How do I create an S3 Bucket?* in the *Amazon Simple Storage Service Console User Guide*. 


Using Lambda with infrastructure as code (IaC)

Lambda offers several ways to deploy your code and create functions. For instance, you can use the Lambda console or the AWS Command Line Interface (AWS CLI) to manually create or update Lambda functions. In addition to these manual options, AWS offers a number of solutions for deploying Lambda functions and serverless applications using infrastructure as code (IaC). With IaC, you can provision and maintain Lambda functions and other AWS resources using code instead of using manual processes and settings.

Most of the time, Lambda functions don’t run in isolation. Instead, they form part of a serverless application with other resources such as databases, queues, and storage. With IaC, you can automate your deployment processes to quickly and repeatably deploy and update whole serverless applications involving many separate AWS resources. This approach speeds up your development cycle, makes configuration management easier, and ensures that your resources are deployed the same way every time.

Topics
- IaC tools for Lambda
- Getting started with IaC for Lambda
- Next steps
- Supported regions for Lambda integration with Application Composer

IaC tools for Lambda

To deploy Lambda functions and serverless applications using IaC, AWS offers a number of different tools and services.

AWS CloudFormation was the first service offered by AWS to create and configure cloud resources. With AWS CloudFormation, you create text templates to define infrastructure and code. As AWS introduced more new services and the complexity of creating AWS CloudFormation templates increased, two further tools were released. AWS SAM is another template-based framework for defining serverless applications. The AWS Cloud Development Kit (AWS CDK) is a code-first approach for defining and provisioning infrastructure using code constructs in many popular programming languages.
With both AWS SAM and the AWS CDK, AWS CloudFormation operates behind the scenes to build and deploy your infrastructure. The following diagram illustrates the relationship between these tools, and the paragraphs after the diagram explain their key features.

- **AWS CloudFormation** - With CloudFormation you model and set up your AWS resources using a YAML or JSON template that describes your resources and their properties. CloudFormation provisions your resources in a safe, repeatable manner, enabling you to frequently build your infrastructure and applications without manual steps. When you change the configuration, CloudFormation determines the right operations to perform to update your stack. CloudFormation can even roll back changes.

- **AWS Serverless Application Model (AWS SAM)** - AWS SAM is an open-source framework for defining serverless applications. AWS SAM templates use a shorthand syntax to define functions, APIs, databases, and event source mappings with just a few lines of text (YAML) per resource. During deployment, AWS SAM transforms and expands the AWS SAM syntax into AWS CloudFormation syntax. Because of this, any CloudFormation syntax can be added to AWS SAM templates. This gives AWS SAM all the power of CloudFormation, but with fewer lines of configuration.

- **AWS Cloud Development Kit (AWS CDK)** - With the AWS CDK, you define your infrastructure using code constructs and provision it through AWS CloudFormation. AWS CDK enables you to model application infrastructure with TypeScript, Python, Java, .NET, and Go (in Developer Preview) using your existing IDE, testing tools, and workflow patterns. You get all the benefits of AWS CloudFormation, including repeatable deployment, easy rollback, and drift detection.
AWS also provides a service called AWS Application Composer to develop IaC templates using a simple graphical interface. With Application Composer, you design an application architecture by dragging, grouping, and connecting AWS services in a visual canvas. Application Composer then creates an AWS SAM template or an AWS CloudFormation template from your design that you can use to deploy your application.

In the section called “Getting started with IaC for Lambda” section below, you use Application Composer to develop a template for a serverless application based on an existing Lambda function.

**Getting started with IaC for Lambda**

In this tutorial, you can get started using IaC with Lambda by creating an AWS SAM template from an existing Lambda function and then building out a serverless application in Application Composer by adding other AWS resources.

If you'd rather start by carrying out an AWS SAM or AWS CloudFormation tutorial to learn how to work with templates without using Application Composer, you'll find links to other resources in the section called “Next steps” section at the end of this page.

As you carry out this tutorial, you'll learn some fundamental concepts, like how AWS resources are specified in AWS SAM. You'll also learn how to use Application Composer to build a serverless application you can deploy using AWS SAM or AWS CloudFormation.

To complete this tutorial, you'll carry out the following steps:

- Create an example Lambda function
- Use the Lambda console to view the AWS SAM template for the function
- Export your function’s configuration to AWS Application Composer and design a simple serverless application based on your function’s configuration
- Save an updated AWS SAM template you can use as a basis to deploy your serverless application

In the section called “Next steps” section, you'll find resources you can use to learn more about AWS SAM and Application Composer. These resources include links to more advanced tutorials that teach you how to deploy a serverless application using AWS SAM.

**Prerequisites**

In this tutorial, you use Application Composer’s local sync feature to save your template and code files to your local build machine. To use this feature, you need a browser that supports the File
System Access API, which allows web applications to read, write, and save files in your local file system. We recommend using either Google Chrome or Microsoft Edge. For more information about the File System Access API, see What is the File System Access API?

Create a Lambda function

In this first step, you create a Lambda function you can use to complete the rest of the tutorial. To keep things simple, you use the Lambda console to create a basic 'Hello world' function using the Python 3.11 runtime.

To create a 'Hello world' Lambda function using the console

1. Open the Lambda console.
2. Choose Create function.
3. Leave Author from scratch selected, and under Basic information, enter LambdaIaCDemo for Function name.
4. For Runtime, select Python 3.11.
5. Choose Create function.

View the AWS SAM template for your function

Before you export your function configuration to Application Composer, use the Lambda console to view your function's current configuration as an AWS SAM template. By following the steps in this section, you'll learn about the anatomy of an AWS SAM template and how to define resources like Lambda functions to start specifying a serverless application.

To view the AWS SAM template for your function

1. Open the Functions page of the Lambda console.
2. Choose the function you just created (LambdaIaCDemo).
3. In the Function overview pane, choose Template.

In place of the diagram representing your function's configuration, you’ll see an AWS SAM template for your function. The template should look like the following.

```yaml
# This AWS SAM template has been generated from your function's configuration. If your function has one or more triggers, note # that the AWS resources associated with these triggers aren't fully
```
# specified in this template and include placeholder values. Open this template
# in AWS Application Composer or your favorite IDE and modify
# it to specify a serverless application with other AWS resources.

AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
      Policies:
        Statement:
          - Effect: Allow
            Action:
              - logs:CreateLogGroup
          - Effect: Allow
            Action:
              - logs:CreateLogStream
              - logs:PutLogEvents
            Resource:
              - arn:aws:logs:us-east-1:123456789012:log-group:/aws/lambda/
            LambdaIaCDemo:*
Let's take a moment to look at the YAML template for your function and understand some key concepts.

The template starts with the declaration `Transform: AWS::Serverless-2016-10-31`. This declaration is required because behind the scenes, AWS SAM templates are deployed through AWS CloudFormation. Using the `Transform` statement identifies the template as an AWS SAM template file.

Following the `Transform` declaration comes the `Resources` section. This is where the AWS resources you want to deploy with your AWS SAM template are defined. AWS SAM templates can contain a combination of AWS SAM resources and AWS CloudFormation resources. This is because during deployment, AWS SAM templates expand to AWS CloudFormation templates, so any valid AWS CloudFormation syntax can be added to an AWS SAM template.

At the moment, there is just one resource defined in the `Resources` section of the template, your Lambda function `LambdaIaCDemo`. To add a Lambda function to an AWS SAM template, you use the `AWS::Serverless::Function` resource type. The `Properties` of a Lambda function resource define the function's runtime, function handler, and other configuration options. The path to your function's source code that AWS SAM should use to deploy the function is also defined here. To learn more about Lambda function resources in AWS SAM, see `AWS::Serverless::Function` in the *AWS SAM Developer Guide*.

As well as the function properties and configurations, the template also specifies an AWS Identity and Access Management (IAM) policy for your function. This policy gives your function permission to write logs to Amazon CloudWatch Logs. When you create a function in the Lambda console, Lambda automatically attaches this policy to your function. To learn more about specifying an IAM policy for a function in an AWS SAM template, see the `policies` property on the `AWS::Serverless::Function` page of the *AWS SAM Developer Guide*.

To learn more about the structure of AWS SAM templates, see [AWS SAM template anatomy](#).

**Use AWS Application Composer to design a serverless application**

To start building out a simple serverless application using your function's AWS SAM template as a starting point, you export your function configuration to Application Composer and activate Application Composer's local sync mode. Local sync automatically saves your function's code and your AWS SAM template to your local build machine and keeps your saved template synced as you add other AWS resources in Application Composer.
To export your function to Application Composer

1. In the Function Overview pane, choose Export to Application Composer.

   To export your function's configuration and code to Application Composer, Lambda creates an Amazon S3 bucket in your account to temporarily store this data.

2. In the dialog box, choose Confirm and create project to accept the default name for this bucket and export your function's configuration and code to Application Composer.

3. (Optional) To choose another name for the Amazon S3 bucket that Lambda creates, enter a new name and choose Confirm and create project. Amazon S3 bucket names must be globally unique and follow the bucket naming rules.

   Selecting Confirm and create project opens the Application Composer console. On the canvas, you'll see your Lambda function.

4. From the Menu dropdown, choose Activate local sync.

5. In the dialog box that opens, choose Select folder and select a folder on your local build machine.

6. Choose Activate to activate local sync.

To export your function to Application Composer, you need permission to use certain API actions. If you're unable to export your function, see the section called “Required permissions” and make sure you have the permissions you need.

Note

Standard Amazon S3 pricing applies for the bucket Lambda creates when you export a function to Application Composer. The objects that Lambda puts into the bucket are automatically deleted after 10 days, but Lambda doesn't delete the bucket itself. To avoid additional charges being added to your AWS account, follow the instructions in Deleting a bucket after you have exported your function to Application Composer. For more information about the Amazon S3 bucket Lambda creates, see the section called “Application Composer”.
To design your serverless application in Application Composer

After activating local sync, changes you make in Application Composer will be reflected in the AWS SAM template saved on your local build machine. You can now drag and drop additional AWS resources onto the Application Composer canvas to build out your application. In this example, you add an Amazon SQS simple queue as a trigger for your Lambda function and a DynamoDB table for the function to write data to.

1. Add an Amazon SQS trigger to your Lambda function by doing the following:
   a. In the search field in the Resources palette, enter SQS.
   b. Drag the SQS Queue resource onto your canvas and position it to the left of your Lambda function.
   c. Choose Details, and for Logical ID enter LambdaIaCQueue.
   d. Choose Save.
   e. Connect your Amazon SQS and Lambda resources by clicking on the Subscription port on the SQS queue card and dragging it to the left hand port on the Lambda function card. The appearance of a line between the two resources indicates a successful connection. Application Composer also displays a message at the bottom of the canvas indicating that the two resources are successfully connected.

2. Add an Amazon DynamoDB table for your Lambda function to write data to by doing the following:
   a. In the search field in the Resources palette, enter DynamoDB.
   b. Drag the DynamoDB Table resource onto your canvas and position it to the right of your Lambda function.
   c. Choose Details, and for Logical ID enter LambdaIaCTable.
   d. Choose Save.
   e. Connect the DynamoDB table to your Lambda function by clicking on the right hand port of the Lambda function card and dragging it to the left hand port on the DynamoDB card.

Now that you’ve added these extra resources, let’s take a look at the updated AWS SAM template Application Composer has created.
To view your updated AWS SAM template

- On the Application Composer canvas, choose Template to switch from the canvas view to the template view.

Your AWS SAM template should now contain the following additional resources and properties:

- An Amazon SQS queue with the identifier LambdaIaCQueue

  LambdaIaCQueue:
  
  Type: AWS::SQS::Queue
  Properties:
  MessageRetentionPeriod: 345600

  When you add an Amazon SQS queue using Application Composer, Application Composer sets the MessageRetentionPeriod property. You can also set the FifoQueue property by selecting Details on the SQS Queue card and checking or unchecking Fifo queue.

  To set other properties for your queue, you can manually edit the template to add them. To learn more about the AWS::SQS::Queue resource and its available properties, see AWS::SQS::Queue in the AWS CloudFormation User Guide.

- An Events property in your Lambda function definition that specifies the Amazon SQS queue as a trigger for the function

  Events:
  
  LambdaIaCQueue:
  
  Type: SQS
  Properties:
  Queue: !GetAtt LambdaIaCQueue.Arn
  BatchSize: 1

  The Events property consists of an event type and a set of properties that depend on the type. To learn about the different AWS services you can configure to trigger a Lambda function and the properties you can set, see EventSource in the AWS SAM Developer Guide.

- A DynamoDB table with the identifier LambdaIaCTable

  LambdaIaCTable:
  
  Type: AWS::DynamoDB::Table
  Properties:
When you add a DynamoDB table using Application Composer, you can set your table's keys by choosing Details on the DynamoDB table card and editing the key values. Application Composer also sets default values for a number of other properties including BillingMode and StreamViewType.

To learn more about these properties and other properties you can add to your AWS SAM template, see `AWS::DynamoDB::Table` in the AWS CloudFormation User Guide.

- A new IAM policy that gives your function permission to perform CRUD operations on the DynamoDB table you added.

```json
  ... - DynamoDBCrudPolicy:
    TableName: !Ref LambdaIaCTable
```

The complete final AWS SAM template should look like the following.

```json
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
```
Architectures:
- x86_64

EventInvokeConfig:
  - MaximumEventAgeInSeconds: 21600
  - MaximumRetryAttempts: 2

EphemeralStorage:
  - Size: 512

RuntimeManagementConfig:
  - UpdateRuntimeOn: Auto

SnapStart:
  - ApplyOn: None

PackageType: Zip

Policies:
- Statement:
  - Effect: Allow
    Action:
      - logs:CreateLogGroup
  - Effect: Allow
    Action:
      - logs:CreateLogStream
      - logs:PutLogEvents
    Resource:
      - arn:aws:logs:us-east-1:594035263019:log-group:/aws/lambda/

LambdaIaCDemo:
- DynamoDBCrudPolicy:
  - TableName: !Ref LambdaIaCTable

Events:
  LambdaIaCQueue:
    - Type: SQS
    - Properties:
      - Queue: !GetAtt LambdaIaCQueue.Arn
      - BatchSize: 1

Environment:
- Variables:
  - LAMBDAIACTABLE_TABLE_NAME: !Ref LambdaIaCTable
  - LAMBDAIACTABLE_TABLE_ARN: !GetAtt LambdaIaCTable.Arn

LambdaIaCQueue:
- Type: AWS::SQS::Queue
- Properties:
  - MessageRetentionPeriod: 345600

LambdaIaCTable:
- Type: AWS::DynamoDB::Table
- Properties:
Deploy your serverless application using AWS SAM (optional)

If you want to use AWS SAM to deploy a serverless application using the template you just created in Application Composer, you first need to install the AWS SAM CLI. To do this, follow the instructions in [Installing the AWS SAM CLI](#).

Before you deploy your application, you also need to update the function code that Application Composer saved along with your template. At the moment, the `lambda_function.py` file that Application Composer saved contains only the basic 'Hello world' code that Lambda provided when you created the function.

To update your function code, copy the following code and paste it into the `lambda_function.py` file Application Composer saved to your local build machine. You specified the directory for Application Composer to save this file to when you activated Local Sync mode.

This code accepts a key value pair in a message from the Amazon SQS queue you created in Application Composer. If both the key and value are strings, the code then uses them to write an item to the DynamoDB table defined in your template.

**Updated Python function code**

```python
import boto3
import os
import json

# define the DynamoDB table that Lambda will connect to
tablename = os.environ['LAMBDAIACTABLE_TABLE_NAME']

# create the DynamoDB resource
dynamo = boto3.client('dynamodb')
```
def lambda_handler(event, context):
    # get the message out of the SQS event
    message = event['Records'][0]['body']
    data = json.loads(message)
    # write event data to DDB table
    if check_message_format(data):
        key = next(iter(data))
        value = data[key]
        dynamo.put_item(
            TableName=tablename,
            Item={
                'id': {'S': key},
                'Value': {'S': value}
            }
        )
    else:
        raise ValueError("Input data not in the correct format")

# check that the event object contains a single key value
# pair that can be written to the database
def check_message_format(message):
    if len(message) != 1:
        return False
    key, value = next(iter(message.items()))
    if not (isinstance(key, str) and isinstance(value, str)):
        return False
    else:
        return True

To deploy your serverless application

To deploy your application using the AWS SAM CLI, carry out the following steps. For your function to build and deploy correctly, Python version 3.11 must be installed on your build machine and on your PATH.

1. Run the following command from the directory in which Application Composer saved your template.yaml and lambda_function.py files.

    sam build
This command gathers the build artifacts for your application and places them in the proper format and location to deploy them.

2. To deploy your application and create the Lambda, Amazon SQS, and DynamoDB resources specified in your AWS SAM template, run the following command.

```
sam deploy --guided
```

Using the `--guided` flag means that AWS SAM will show you prompts to guide you through the deployment process. For this deployment, accept the default options by pressing Enter.

During the deployment process, AWS SAM creates the following resources in your AWS account:

- An AWS CloudFormation stack named `sam-app`
- A Lambda function with the name format `sam-app-LambdaIaCDemo-99VXPpYQVv1M`
- An Amazon SQS queue with the name format `sam-app-LambdaIaCQueue-xL87VeKsGiIo`
- A DynamoDB table with the name format `sam-app-LambdaIaCTable-CN0S66C0VLNV`

AWS SAM also creates the necessary IAM roles and policies so that your Lambda function can read messages from the Amazon SQS queue and perform CRUD operations on the DynamoDB table.

To learn more about using AWS SAM to deploy serverless applications, see the resources in the section called “Next steps” section.

**Testing your deployed application (optional)**

To confirm that your serverless application deployed correctly, send a message to your Amazon SQS queue containing a key value pair and check that Lambda writes an item into your DynamoDB table using these values.

**To test your serverless application**

1. Open the Queues page of the Amazon SQS console and select the queue that AWS SAM created from your template. The name has the format `sam-app-LambdaIaCQueue-xL87VeKsGiIo`.

2. Choose **Send and receive messages** and paste the following JSON into the **Message body** in the **Send message** section.

```json
{
    "key": "value"
}
```
3. **Choose Send message.**

Sending your message to the queue causes Lambda to invoke your function through the event source mapping defined in your AWS SAM template. To confirm that Lambda has invoked your function as expected, confirm that an item has been added to your DynamoDB table.

4. Open the **Tables** page of the DynamoDB console and select your table. The name has the format `sam-app-LambdaIaCTable-CN0S66C0VLNV`.

5. Choose **Explore table items**. In the **Items returned** pane, you should see an item with the id `myKey` and the Value `myValue`.

**Next steps**

To learn more about using Application Composer with AWS SAM and AWS CloudFormation, start with **Using Application Composer with AWS CloudFormation and AWS SAM**.

For a guided tutorial that uses AWS SAM to deploy a serverless application designed in Application Composer, we also recommend you carry out the **AWS Application Composer tutorial** in the **AWS Serverless Patterns Workshop**.

AWS SAM provides a command line interface (CLI) that you can use with AWS SAM templates and supported third-party integrations to build and run your serverless applications. With the AWS SAM CLI, you can build and deploy your application, perform local testing and debugging, configure CI/CD pipelines, and more. To learn more about using the AWS SAM CLI, see **Getting started with AWS SAM** in the **AWS Serverless Application Model Developer Guide**.

To learn how to deploy a serverless application with an AWS SAM template using the AWS CloudFormation console, start with **Using the AWS CloudFormation console** in the **AWS CloudFormation User Guide**.

**Supported regions for Lambda integration with Application Composer**

Lambda integration with Application Composer is supported in the following AWS Regions:

- US East (N. Virginia)
- US East (Ohio)
- US West (N. California)
- US West (Oregon)
- Africa (Cape Town)
- Asia Pacific (Hong Kong)
- Asia Pacific (Hyderabad)
- Asia Pacific (Jakarta)
- Asia Pacific (Melbourne)
- Asia Pacific (Mumbai)
- Asia Pacific (Osaka)
- Asia Pacific (Seoul)
- Asia Pacific (Singapore)
- Asia Pacific (Sydney)
- Asia Pacific (Tokyo)
- Canada (Central)
- Europe (Frankfurt)
- Europe (Zurich)
- Europe (Ireland)
- Europe (London)
- Europe (Stockholm)
- Middle East (UAE)
Private networking with VPC

Amazon Virtual Private Cloud (Amazon VPC) is a virtual network in the AWS cloud, dedicated to your AWS account. You can use Amazon VPC to create a private network for resources such as databases, cache instances, or internal services. For more information about Amazon VPC, see What is Amazon VPC?

A Lambda function always runs inside a VPC owned by the Lambda service. Lambda applies network access and security rules to this VPC and Lambda maintains and monitors the VPC automatically. If your Lambda function needs to access the resources in your account VPC, configure the function to access the VPC. Lambda provides managed resources named Hyperplane ENIs, which your Lambda function uses to connect from the Lambda VPC to an ENI (Elastic network interface) in your account VPC.

There's no additional charge for using a VPC or a Hyperplane ENI. There are charges for some VPC components, such as NAT gateways. For more information, see Amazon VPC Pricing.

Topics
- VPC network elements
- Connecting Lambda functions to your VPC
- Shared subnets
- Lambda Hyperplane ENIs
- Connections
- IPv6 support
- Security
- Observability

VPC network elements

Amazon VPC networks includes the following network elements:

- Elastic network interface – elastic network interface is a logical networking component in a VPC that represents a virtual network card.
- Subnet – A range of IP addresses in your VPC. You can add AWS resources to a specified subnet. Use a public subnet for resources that must connect to the internet, and a private subnet for resources that don't connect to the internet.
• Security group – use security groups to control access to the AWS resources in each subnet.
• Access control list (ACL) – use a network ACL to provide additional security in a subnet. The default subnet ACL allows all inbound and outbound traffic.
• Route table – contains a set of routes that AWS uses to direct the network traffic for your VPC. You can explicitly associate a subnet with a particular route table. By default, the subnet is associated with the main route table.
• Route – each route in a route table specifies a range of IP addresses and the destination where Lambda sends the traffic for that range. The route also specifies a target, which is the gateway, network interface, or connection through which to send the traffic.
• NAT gateway – An AWS Network Address Translation (NAT) service that controls access from a private VPC private subnet to the Internet.
• VPC endpoints – You can use an Amazon VPC endpoint to create private connectivity to services hosted in AWS, without requiring access over the internet or through a NAT device, VPN connection, or AWS Direct Connect connection. For more information, see AWS PrivateLink and VPC endpoints.

Tip
To configure your Lambda function to access a VPC and subnet, you can use the Lambda Console or the API. Refer to the VpcConfig section in CreateFunction to configure your function. See Configuring VPC access (console) and Configuring VPC access (API) for detailed steps.

For more information about Amazon VPC networking definitions, see How Amazon VPC works in the Amazon VPC Developer Guide and the Amazon VPC FAQs.

Connecting Lambda functions to your VPC

A Lambda function always runs inside a VPC owned by the Lambda service. By default, a Lambda function isn't connected to VPCs in your account. When you connect a function to a VPC in your account, the function can't access the internet unless your VPC provides access.

Lambda accesses resources in your VPC using a Hyperplane ENI. Hyperplane ENIs provide NAT capabilities from the Lambda VPC to your account VPC using VPC-to-VPC NAT (V2N). V2N provides connectivity from the Lambda VPC to your account VPC, but not in the other direction.
When you create a Lambda function (or update its VPC settings), Lambda allocates a Hyperplane ENI for each subnet in your function's VPC configuration. Multiple Lambda functions can share a network interface, if the functions share the same subnet and security group.

To connect to another AWS service, you can use VPC endpoints for private communications between your VPC and supported AWS services. An alternative approach is to use a NAT gateway to route outbound traffic to another AWS service.

To give your function access to the internet, route outbound traffic to a NAT gateway in a public subnet. The NAT gateway has a public IP address and can connect to the internet through the VPC's internet gateway.

For information about how to configure Lambda VPC networking, see ??? and ???.

Shared subnets

VPC sharing allows multiple AWS accounts to create their application resources, such as Amazon EC2 instances and Lambda functions, in shared, centrally-managed virtual private clouds (VPCs). In this model, the account that owns the VPC (owner) shares one or more subnets with other accounts (participants) that belong to the same AWS Organization.

To access private resources, connect your function to a private shared subnet in your VPC. The subnet owner must share a subnet with you before you can connect a function to it. The subnet owner can also unshare the subnet a later time, thereby removing connectivity. For details on how to share, unshare, and manage VPC resources in shared subnets, see How to share your VPC with other accounts in the Amazon VPC guide.

Lambda Hyperplane ENIs

The Hyperplane ENI is a managed network resource that the Lambda service creates and manages. Multiple execution environments in the Lambda VPC can use a Hyperplane ENI to securely access resources inside of VPCs in your account. Hyperplane ENIs provide NAT capabilities from the Lambda VPC to your account VPC.

For each subnet, Lambda creates a network interface for each unique set of security groups. Functions in the account that share the same subnet and security group combination will use the same network interfaces. Connections made through the Hyperplane layer are automatically tracked, even if the security group configuration does not otherwise require tracking. Inbound packets from the VPC that don't correspond to established connections are dropped at the
Hyperplane layer. For more information, see Security group connection tracking in the Amazon EC2 User Guide for Linux Instances.

Because the functions in your account share the ENI resources, the ENI lifecycle is more complex than other Lambda resources. The following sections describe the ENI lifecycle.

**ENI lifecycle**

- Creating ENIs
- Managing ENIs
- Deleting ENIs

**Creating ENIs**

Lambda may create Hyperplane ENI resources for a newly created VPC-enabled function or for a VPC configuration change to an existing function. The function remains in pending state while Lambda creates the required resources. When the Hyperplane ENI is ready, the function transitions to active state and the ENI becomes available for use. Lambda can require several minutes to create a Hyperplane ENI.

For a newly created VPC-enabled function, any invocations or other API actions that operate on the function fail until the function state transitions to active.

For a VPC configuration change to an existing function, any function invocations continue to use the Hyperplane ENI associated with the old subnet and security group configuration until the function state transitions to active.

If a Lambda function remains idle for consecutive weeks, Lambda reclaims the unused Hyperplane ENIs and sets the function state to idle. The next invocation causes Lambda to reactivate the idle function. The invocation fails, and the function enters pending state until Lambda completes the creation or allocation of a Hyperplane ENI.

For more information about function states, see Lambda function states.

**Managing ENIs**

Lambda uses permissions in your function's execution role to create and manage network interfaces. Lambda creates a Hyperplane ENI when you define a unique subnet plus security group combination for a VPC-enabled function in an account. Lambda reuses the Hyperplane ENI...
for other VPC-enabled functions in your account that use the same subnet and security group combination.

There is no quota on the number of Lambda functions that can use the same Hyperplane ENI. However, each Hyperplane ENI supports up to 65,000 connections/ports. If the number of connections exceeds 65,000, Lambda creates a new Hyperplane ENI to provide additional connections.

When you update your function configuration to access a different VPC, Lambda terminates connectivity to the Hyperplane ENI in the previous VPC. The process to update the connectivity to a new VPC can take several minutes. During this time, invocations to the function continue to use the previous VPC. After the update is complete, new invocations start using the Hyperplane ENI in the new VPC. At this point, the Lambda function is no longer connected to the previous VPC.

Deleting ENIs

When you update a function to remove its VPC configuration, Lambda requires up to 20 minutes to delete the attached Hyperplane ENI. Lambda only deletes the ENI if no other function (or published function version) is using that Hyperplane ENI.

Lambda relies on permissions in the function execution role to delete the Hyperplane ENI. If you delete the execution role before Lambda deletes the Hyperplane ENI, Lambda won't be able to delete the Hyperplane ENI. You can manually perform the deletion.

Lambda doesn't delete network interfaces that are in use by functions or function versions in your account. You can use the Lambda ENI Finder to identify the functions or function versions that are using a Hyperplane ENI. For any functions or function versions that you no longer need, you can remove the VPC configuration so that Lambda deletes the Hyperplane ENI.

Connections

Lambda supports two types of connections: TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).

When you create a VPC, Lambda automatically creates a set of DHCP options and associates them with the VPC. You can configure your own DHCP options set for your VPC. For more details, refer to Amazon VPC DHCP options.

Amazon provides a DNS server (the Amazon Route 53 resolver) for your VPC. For more information, see DNS support for your VPC.
IPv6 support

Lambda supports inbound connections to Lambda's public dual-stack endpoints, and outbound connections to dual-stack VPC subnets over IPv6.

Inbound

To invoke your function over IPv6, use Lambda's public dual-stack endpoints. Dual-stack endpoints support both IPv4 and IPv6. Lambda dual-stack endpoints use the following syntax:

```
protocol://lambda.us-east-1.api.aws
```

You can also use Lambda function URLs to invoke functions over IPv6. Function URL endpoints have the following format:

```
https://url-id.lambda-url.us-east-1.on.aws
```

Outbound

Your function can connect to resources in dual-stack VPC subnets over IPv6. This option is turned off by default. To allow outbound IPv6 traffic, use the console or the `--vpc-config Ipv6AllowedForDualStack=true` option with the `create-function` or `update-function-configuration` command.

Note

To allow outbound IPv6 traffic in a VPC, all of the subnets that are connected to the function must be dual-stack subnets. Lambda doesn't support outbound IPv6 connections for IPv6-only subnets in a VPC, outbound IPv6 connections for functions that are not connected to a VPC, or inbound IPv6 connections using VPC endpoints (AWS PrivateLink).

You can update your function code to explicitly connect to subnet resources over IPv6. The following Python example opens a socket and connects to an IPv6 server.

Example — Connect to IPv6 server

```python
def connect_to_server(event, context):
```
server_address = event['host']
server_port = event['port']
message = event['message']
run_connect_to_server(server_address, server_port, message)

def run_connect_to_server(server_address, server_port, message):
    sock = socket.socket(socket.AF_INET6, socket.SOCK_STREAM, 0)
    try:
        # Send data
        sock.connect((server_address, int(server_port), 0, 0))
        sock.sendall(message.encode())
        BUFF_SIZE = 4096
        data = b''
        while True:
            segment = sock.recv(BUFF_SIZE)
            data += segment
            # Either 0 or end of data
            if len(segment) < BUFF_SIZE:
                break
        return data
    finally:
        sock.close()

Security

AWS provides security groups and network ACLs to increase security in your VPC. Security groups control inbound and outbound traffic for your resources, and network ACLs control inbound and outbound traffic for your subnets. Security groups provide enough access control for most subnets. You can use network ACLs if you want an additional layer of security for your VPC. For more information, see Internetwork traffic privacy in Amazon VPC. Every subnet that you create is automatically associated with the VPC's default network ACL. You can change the association, and you can change the contents of the default network ACL.

For general security best practices, see VPC security best practices. For details on how you can use IAM to manage access to the Lambda API and resources, see AWS Lambda permissions.

You can use Lambda-specific condition keys for VPC settings to provide additional permission controls for your Lambda functions. For more information about VPC condition keys, see Using IAM condition keys for VPC settings.
Note

Lambda functions can be invoked from the public internet or AWS PrivateLink endpoints. You can access your Function URLs through the public Internet only. While Lambda functions do support AWS PrivateLink, Function URLs do not.

Observability

You can use VPC Flow Logs to capture information about the IP traffic going to and from network interfaces in your VPC. You can publish Flow log data to Amazon CloudWatch Logs or Amazon S3. After you've created a flow log, you can retrieve and view its data in the chosen destination.

Note: when you attach a function to a VPC, the CloudWatch log messages do not use the VPC routes. Lambda sends them using the regular routing for logs.
Lambda console

You can use the Lambda console to configure applications, functions, code signing configurations, and layers. This page provides an explanation for how to edit code using the console editor.

Topics
- Applications
- Functions
- Code signing
- Layers
- Edit code using the console editor

Applications

The Applications page displays a list of applications that were deployed using AWS CloudFormation or other tools, including the AWS Serverless Application Model (AWS SAM). Filter to find applications based on keywords.

Functions

The functions page shows you a list of functions defined for your account in this region. The initial console flow to create a function depends on whether the function uses a container image or .zip file archive for the deployment package. Many of the optional configuration tasks are common to both types of function.

The console provides a code editor for your convenience.

Code signing

You can attach a code signing configuration to a function. With code signing, you can ensure that the code has been signed by an approved source and has not been altered since signing, and that the code signature has not expired or been revoked.

Layers

Create layers to separate your .zip archive function code from its dependencies. A layer is a .zip file archive that contains libraries, a custom runtime, or other dependencies. With layers, you can use libraries in your function without needing to include them in your deployment package.
Edit code using the console editor

You can use the code editor in the Lambda console to write, test, and view the execution results of your Lambda function code. The code editor supports languages that do not require compiling, such as Node.js and Python. The code editor supports only .zip file archive deployment packages, and the size of the deployment package must be less than 3 MB.

The code editor includes the *menu bar*, *windows*, and the *editor pane*.

For a list of what the commands do, see the [menu bar commands reference](#) in the *AWS Cloud9 User Guide*. Note that some of the commands listed in that reference are not available in the code editor.

Topics

- Working with files and folders
- Working with code
- Working in fullscreen mode
- Working with preferences

Working with files and folders

You can use the **Environment** window in the code editor to create, open, and manage files for your function.
To show or hide the Environment window, choose the Environment button. If the Environment button is not visible, choose Window, Environment on the menu bar.

To open a single file and show its contents in the editor pane, double-click the file in the Environment window.

To open multiple files and show their contents in the editor pane, choose the files in the Environment window. Right-click the selection, and then choose Open.

To create a new file, do one of the following:

- In the Environment window, right-click the folder where you want the new file to go, and then choose New File. Enter the file's name and extension, and then press Enter.
- Choose File, New File on the menu bar. When you're ready to save the file, choose File, Save or File, Save As on the menu bar. Then use the Save As dialog box that displays to name the file and choose where to save it.
- In the tab buttons bar in the editor pane, choose the + button, and then choose New File. When you're ready to save the file, choose File, Save or File, Save As on the menu bar. Then use the Save As dialog box that displays to name the file and choose where to save it.
To create a new folder, right-click the folder in the Environment window where you want the new folder to go, and then choose New Folder. Enter the folder's name, and then press Enter.

To save a file, with the file open and its contents visible in the editor pane, choose File, Save on the menu bar.

To rename a file or folder, right-click the file or folder in the Environment window. Enter the replacement name, and then press Enter.

To delete files or folders, choose the files or folders in the Environment window. Right-click the selection, and then choose Delete. Then confirm the deletion by choosing Yes (for a single selection) or Yes to All.

To cut, copy, paste, or duplicate files or folders, choose the files or folders in the Environment window. Right-click the selection, and then choose Cut, Copy, Paste, or Duplicate, respectively.

To collapse folders, choose the gear icon in the Environment window, and then choose Collapse All Folders.

To show or hide hidden files, choose the gear icon in the Environment window, and then choose Show Hidden Files.
To see environment variables that are configured for the function, do the following:

1. Choose the **Code** tab.
2. Choose the **Environment Variables** tab.
3. Choose **Tools, Show Environment Variables**.

Environment variables remain encrypted when listed in the console code editor. If you enabled encryption helpers for encryption in transit, then those settings remain unchanged. For more information, see [Securing environment variables](#).

The environment variables list is read-only and is available only on the Lambda console. This file is not included when you download the function's .zip file archive, and you can't add environment variables by uploading this file.

**Working with code**

Use the editor pane in the code editor to view and write code.

![Code editor screenshot](image)

**Working with tab buttons**

Use the *tab buttons bar* to select, view, and create files.

![Tab buttons screenshot](image)
To display an open file's contents, do one of the following:

- Choose the file's tab.
- Choose the drop-down menu button in the tab buttons bar, and then choose the file's name.

To close an open file, do one of the following:

- Choose the X icon in the file's tab.
- Choose the file's tab. Then choose the drop-down menu button in the tab buttons bar, and choose Close Pane.

To close multiple open files, choose the drop-down menu in the tab buttons bar, and then choose Close All Tabs in All Panes or Close All But Current Tab as needed.

To create a new file, choose the + button in the tab buttons bar, and then choose New File. When you're ready to save the file, choose File, Save or File, Save As on the menu bar. Then use the Save As dialog box that displays to name the file and choose where to save it.

Working with the status bar

Use the status bar to move quickly to a line in the active file and to change how code is displayed.
To move quickly to a line in the active file, choose the line selector, enter the line number to go to, and then press Enter.

To change the code color scheme in the active file, choose the code color scheme selector, and then choose the new code color scheme.
To change in the active file whether soft tabs or spaces are used, the tab size, or whether to convert to spaces or tabs, choose the spaces and tabs selector, and then choose the new settings.

![Spaces 4](image)

To change for all files whether to show or hide invisible characters or the gutter, auto-pair brackets or quotes, wrap lines, or the font size, choose the gear icon, and then choose the new settings.

![Settings](image)

Working in fullscreen mode

You can expand the code editor to get more room to work with your code.

To expand the code editor to the edges of the web browser window, choose the **Toggle fullscreen** button in the menu bar.
To shrink the code editor to its original size, choose the **Toggle fullscreen** button again.

In fullscreen mode, additional options are displayed on the menu bar: **Save** and **Test**. Choosing **Save** saves the function code. Choosing **Test** or **Configure Events** enables you to create or edit the function's test events.

**Working with preferences**

You can change various code editor settings such as which coding hints and warnings are displayed, code folding behaviors, code autocompletion behaviors, and much more.

To change code editor settings, choose the **Preferences** gear icon in the menu bar.

For a list of what the settings do, see the following references in the *AWS Cloud9 User Guide*.

- [Project settings that you can change](#)
- [User setting changes you can make](#)

Note that some of the settings listed in those references are not available in the code editor.
Lambda instruction set architectures (ARM/x86)

The instruction set architecture of a Lambda function determines the type of computer processor that Lambda uses to run the function. Lambda provides a choice of instruction set architectures:

- arm64 – 64-bit ARM architecture, for the AWS Graviton2 processor.
- x86_64 – 64-bit x86 architecture, for x86-based processors.

**Note**

The arm64 architecture is available in most AWS Regions. For more information, see [AWS Lambda Pricing](#). In the memory prices table, choose the Arm Price tab, and then open the Region dropdown list to see which AWS Regions support arm64 with Lambda.

For an example of how to create a function with arm64 architecture, see [AWS Lambda Functions Powered by AWS Graviton2 Processor](#).

**Advantages of using arm64 architecture**

Lambda functions that use arm64 architecture (AWS Graviton2 processor) can achieve significantly better price and performance than the equivalent function running on x86_64 architecture. Consider using arm64 for compute-intensive applications such as high-performance computing, video encoding, and simulation workloads.

The Graviton2 CPU uses the Neoverse N1 core and supports Armv8.2 (including CRC and crypto extensions) plus several other architectural extensions.

Graviton2 reduces memory read time by providing a larger L2 cache per vCPU, which improves the latency performance of web and mobile backends, microservices, and data processing systems.
Graviton2 also provides improved encryption performance and supports instruction sets that improve the latency of CPU-based machine learning inference.

For more information about AWS Graviton2, see AWS Graviton Processor.

Requirements for migration to arm64 architecture

When you select a Lambda function to migrate to arm64 architecture, to ensure a smooth migration, make sure that your function meets the following requirements:

- The function currently uses a Lambda Amazon Linux 2 runtime.
- The deployment package contains only open-source components and source code that you control, so that you can make any necessary updates for the migration.
- If the function code includes third-party dependencies, each library or package provides an arm64 version.

Function code compatibility with arm64 architecture

Your Lambda function code must be compatible with the instruction set architecture of the function. Before you migrate a function to arm64 architecture, note the following points about the current function code:

- If you added your function code using the embedded code editor, your code probably runs on either architecture without modification.
- If you uploaded your function code, you must upload new code that is compatible with your target architecture.
- If your function uses layers, you must check each layer to ensure that it is compatible with the new architecture. If a layer is not compatible, edit the function to replace the current layer version with a compatible layer version.
- If your function uses Lambda extensions, you must check each extension to ensure that it is compatible with the new architecture.
- If your function uses a container image deployment package type, you must create a new container image that is compatible with the architecture of the function.

How to migrate to arm64 architecture
To migrate a Lambda function to the arm64 architecture, we recommend following these steps:

1. Build the list of dependencies for your application or workload. Common dependencies include:
   • All the libraries and packages that the function uses.
   • The tools that you use to build, deploy, and test the function, such as compilers, test suites, continuous integration and continuous delivery (CI/CD) pipelines, provisioning tools, and scripts.
   • The Lambda extensions and third-party tools that you use to monitor the function in production.
2. For each of the dependencies, check the version, and then check whether arm64 versions are available.
3. Build an environment to migrate your application.
4. Bootstrap the application.
5. Test and debug the application.
6. Test the performance of the arm64 function. Compare the performance with the x86_64 version.
7. Update your infrastructure pipeline to support arm64 Lambda functions.
8. Stage your deployment to production.
   
   For example, use alias routing configuration to split traffic between the x86 and arm64 versions of the function, and compare the performance and latency.

For more information about how to create a code environment for arm64 architecture, including language-specific information for Java, Go, .NET, and Python, see the Getting started with AWS Graviton GitHub repository.

**Configuring the instruction set architecture**

You can configure the instruction set architecture for new and existing Lambda functions using the Lambda console, AWS SDKs, AWS Command Line Interface (AWS CLI), or AWS CloudFormation. Follow these steps to change the instruction set architecture for an existing Lambda function from the console.

1. Open the Functions page of the Lambda console.
2. Choose the name of the function that you want to configure the instruction set architecture for.
3. On the main **Code** tab, for the **Runtime settings** section, choose **Edit**.

4. Under **Architecture**, choose the instruction set architecture you want your function to use.

5. Choose **Save**.

---

**Note**

All Amazon Linux 2 runtimes support both x86_64 and ARM CPU architectures. Runtimes that do **not** use Amazon Linux 2, such as Go 1.x, do not support the arm64 architecture. To use arm64 architecture with Go 1.x, you can run your function in a provided.al2 runtime. For more information, see the deployment instructions for .zip packages and container images.
Additional Lambda features

Lambda provides a management console and API for managing and invoking functions. It provides runtimes that support a standard set of features so that you can easily switch between languages and frameworks, depending on your needs. In addition to functions, you can also create versions, aliases, layers, and custom runtimes.

Advanced features

- Scaling
- Concurrency controls
- Function URLs
- Asynchronous invocation
- Event source mappings
- Destinations
- Function blueprints
- Testing and deployment tools
- Application templates

Scaling

Lambda manages the infrastructure that runs your code, and scales automatically in response to incoming requests. When your function is invoked more quickly than a single instance of your function can process events, Lambda scales up by running additional instances. When traffic subsides, inactive instances are frozen or stopped. You pay only for the time that your function is initializing or processing events.

For more information, see Lambda function scaling.

Concurrency controls

Use concurrency settings to ensure that your production applications are highly available and highly responsive.

To prevent a function from using too much concurrency, and to reserve a portion of your account's available concurrency for a function, use reserved concurrency. Reserved concurrency splits the pool
of available concurrency into subsets. A function with reserved concurrency only uses concurrency from its dedicated pool.

To enable functions to scale without fluctuations in latency, use provisioned concurrency. For functions that take a long time to initialize, or that require extremely low latency for all invocations, provisioned concurrency enables you to pre-initialize instances of your function and keep them running at all times. Lambda integrates with Application Auto Scaling to support autoscaling for provisioned concurrency based on utilization.

For more information, see Configuring reserved concurrency.

**Function URLs**

Lambda offers built-in HTTP(S) endpoint support through function URLs. With function URLs, you can assign a dedicated HTTP endpoint to your Lambda function. When your function URL is configured, you can use it to invoke your function through a web browser, curl, Postman, or any HTTP client.

You can add a function URL to an existing function, or create a new function with a function URL. For more information, see Invoking Lambda function URLs.

**Asynchronous invocation**

When you invoke a function, you can choose to invoke it synchronously or asynchronously. With synchronous invocation, you wait for the function to process the event and return a response. With asynchronous invocation, Lambda queues the event for processing and returns a response immediately.

![Asynchronous Invocation Diagram](image-url)
For asynchronous invocations, Lambda handles retries if the function returns an error or is throttled. To customize this behavior, you can configure error handling settings on a function, version, or alias. You can also configure Lambda to send events that failed processing to a dead-letter queue, or to send a record of any invocation to a destination.

For more information, see Asynchronous invocation.

**Event source mappings**

To process items from a stream or queue, you can create an event source mapping. An event source mapping is a resource in Lambda that reads items from an Amazon Simple Queue Service (Amazon SQS) queue, an Amazon Kinesis stream, or an Amazon DynamoDB stream, and sends the items to your function in batches. Each event that your function processes can contain hundreds or thousands of items.

Event source mappings maintain a local queue of unprocessed items and handle retries if the function returns an error or is throttled. You can configure an event source mapping to customize
batching behavior and error handling, or to send a record of items that fail processing to a destination.

For more information, see Lambda event source mappings.

Destinations

A destination is an AWS resource that receives invocation records for a function. For asynchronous invocation, you can configure Lambda to send invocation records to a queue, topic, function, or event bus. You can configure separate destinations for successful invocations and events that failed processing. The invocation record contains details about the event, the function's response, and the reason that the record was sent.

For event source mappings that read from streams, you can configure Lambda to send a record of batches that failed processing to a queue or topic. A failure record for an event source mapping contains metadata about the batch, and it points to the items in the stream.
For more information, see Configuring destinations for asynchronous invocation and the error handling sections of Using AWS Lambda with Amazon DynamoDB and Using AWS Lambda with Amazon Kinesis.

Function blueprints

When you create a function in the Lambda console, you can choose to start from scratch, use a blueprint, or use a container image. A blueprint provides sample code that shows how to use Lambda with an AWS service or a popular third-party application. Blueprints include sample code and function configuration presets for Node.js and Python runtimes.

Blueprints are provided for use under the Amazon Software License. They are available only in the Lambda console.

Testing and deployment tools

Lambda supports deploying code as is or as container images. You can use AWS services and popular community tools like the Docker command line interface (CLI) to author, build, and deploy your Lambda functions. To set up the Docker CLI, see Get Docker on the Docker Docs website. For an introduction to using Docker with AWS, see Getting started with Amazon ECR using the AWS CLI in the Amazon Elastic Container Registry User Guide.

The AWS CLI and AWS SAM CLI are command line tools for managing Lambda application stacks. In addition to commands for managing application stacks with the AWS CloudFormation API, the AWS CLI supports higher-level commands that simplify tasks such as uploading deployment packages and updating templates. The AWS SAM CLI provides additional functionality, including validating templates, testing locally, and integrating with CI/CD systems.

- Installing the AWS SAM CLI
- Testing and debugging serverless applications with AWS SAM
- Deploying serverless applications using CI/CD systems with AWS SAM

Application templates

You can use the Lambda console to create an application with a continuous delivery pipeline. Application templates in the Lambda console include code for one or more functions, an application template that defines functions and supporting AWS resources, and an infrastructure
template that defines an AWS CodePipeline pipeline. The pipeline has build and deploy stages that run every time you push changes to the included Git repository.

Application templates are provided for use under the MIT No Attribution license. They are available only in the Lambda console.

For more information, see Creating an application with continuous delivery in the Lambda console.

Learn how to build serverless solutions

🧩 Tip
To learn how to build serverless solutions, check out the Serverless Developer Guide.
Lambda runtimes

Lambda supports multiple languages through the use of runtimes. A runtime provides a language-specific environment that relays invocation events, context information, and responses between Lambda and the function. You can use runtimes that Lambda provides, or build your own.

Each major programming language release has a separate runtime, with a unique runtime identifier, such as nodejs20.x or python3.12. To configure a function to use a new major language version, you need to change the runtime identifier. Since AWS Lambda cannot guarantee backward compatibility between major versions, this is a customer-driven operation.

For a function defined as a container image, you choose a runtime and the Linux distribution when you create the container image. To change the runtime, you create a new container image.

When you use a .zip file archive for the deployment package, you choose a runtime when you create the function. To change the runtime, you can update your function's configuration. The runtime is paired with one of the Amazon Linux distributions. The underlying execution environment provides additional libraries and environment variables that you can access from your function code.

Lambda invokes your function in an execution environment. The execution environment provides a secure and isolated runtime environment that manages the resources required to run your function. Lambda re-uses the execution environment from a previous invocation if one is available, or it can create a new execution environment.

To use other languages in Lambda, such as Go or Rust, use an OS-only runtime. The Lambda execution environment provides a runtime interface for getting invocation events and sending responses. You can deploy other languages by implementing a custom runtime alongside your function code, or in a layer.

The following table lists the supported Lambda runtimes and projected deprecation dates. After a runtime is deprecated, you're still able to create and update functions for a limited period. For more information, see the section called “Runtime support after deprecation”. The table provides the currently forecasted dates for runtime deprecation. These dates are provided for planning purposes and are subject to change.
## Supported Runtimes

<table>
<thead>
<tr>
<th>Name</th>
<th>Identifier</th>
<th>SDK</th>
<th>Operating System</th>
<th>Deprecation date</th>
<th>Block function create</th>
<th>Block function update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node.js 20</td>
<td>nodejs20x</td>
<td>3.362.0</td>
<td>Amazon Linux 2023</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Node.js 18</td>
<td>nodejs18x</td>
<td>3.362.0</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Node.js 16</td>
<td>nodejs16x</td>
<td>2.1374.0</td>
<td>Amazon Linux 2</td>
<td>Jun 12, 2024</td>
<td>Jul 15, 2024</td>
<td>Aug 15, 2024</td>
</tr>
<tr>
<td>Python 3.12</td>
<td>python3.12</td>
<td>boto3-1.2.8.83 botocore-1.31.83</td>
<td>Amazon Linux 2023</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python 3.11</td>
<td>python3.11</td>
<td>boto3-1.2.8.83 botocore-1.31.83</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python 3.10</td>
<td>python3.10</td>
<td>boto3-1.2.6.90 botocore-1.29.90</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python 3.9</td>
<td>python3.9</td>
<td>boto3-1.2.6.90 botocore-1.29.90</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python 3.8</td>
<td>python3.8</td>
<td>boto3-1.2.6.90</td>
<td>Amazon Linux 2</td>
<td>Oct 14, 2024</td>
<td>Nov 13, 2024</td>
<td>Jan 7, 2025</td>
</tr>
<tr>
<td>Name</td>
<td>Identifier</td>
<td>SDK</td>
<td>Operating system</td>
<td>Deprecation date</td>
<td>Block function create</td>
<td>Block function update</td>
</tr>
<tr>
<td>---------------</td>
<td>------------</td>
<td>-------------</td>
<td>------------------</td>
<td>------------------</td>
<td>-----------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>botocore-1.29.90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java 21</td>
<td>java21</td>
<td></td>
<td>Amazon Linux 2023</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java 17</td>
<td>java17</td>
<td></td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java 11</td>
<td>java11</td>
<td></td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java 8</td>
<td>java8.al2:</td>
<td></td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java 8</td>
<td>java8</td>
<td></td>
<td>Amazon Linux</td>
<td>Jan 8, 2023</td>
<td>Feb 8, 2024</td>
<td>Mar 12, 2024</td>
</tr>
<tr>
<td>.NET 7 (container only)</td>
<td>dotnet7</td>
<td></td>
<td>Amazon Linux 2</td>
<td>May 14, 2024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NET 6</td>
<td>dotnet6</td>
<td></td>
<td>Amazon Linux 2</td>
<td>Nov 12, 2024</td>
<td>Jan 11, 2024</td>
<td>Feb 11, 2024</td>
</tr>
<tr>
<td>Go 1.x</td>
<td>go1.x</td>
<td></td>
<td>Amazon Linux</td>
<td>Jan 8, 2024</td>
<td>Feb 8, 2024</td>
<td>Mar 12, 2024</td>
</tr>
<tr>
<td>Ruby 3.2</td>
<td>ruby3.2</td>
<td>3.1.0</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS-only Runtime</td>
<td>provided</td>
<td></td>
<td>Amazon Linux 2023</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Identifier</td>
<td>SDK</td>
<td>Operating system</td>
<td>Deprecation date</td>
<td>Block function create</td>
<td>Block function update</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------</td>
<td>--------------</td>
<td>------------------</td>
<td>------------------</td>
<td>------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>OS-only Runtime</td>
<td>provided a12</td>
<td>Amazon</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS-only Runtime</td>
<td>provided</td>
<td>Amazon</td>
<td>Amazon Linux</td>
<td>Jan 8, 2024</td>
<td>Feb 8, 2024</td>
<td>Mar 12, 2024</td>
</tr>
</tbody>
</table>

<i>Note</i>

For new regions, Lambda will not support runtimes that are set to be deprecated within the next 6 months.

Lambda keeps managed runtimes and their corresponding container base images up to date with patches and support for minor version releases. For more information see [Lambda runtime updates](#).

Lambda continues to support the Go programming language after deprecation of the Go 1.x runtime. For more information, see [Migrating AWS Lambda functions from the Go1.x runtime to the custom runtime on Amazon Linux 2](#) on the [AWS Compute Blog](#).

All Lambda runtimes that use the Amazon Linux 2 or Amazon Linux 2023 operating systems support both x86_64 and arm64 architectures. Runtimes that use Amazon Linux support only x86_64.

**New runtime releases**

Lambda provides managed runtimes for new language versions only when the release reaches the long-term support (LTS) phase of the language's release cycle. For example, for the [Node.js release cycle](#), when the release reaches the Active LTS phase.

Before the release reaches the long-term support phase, it remains in development and can still be subject to breaking changes. Lambda applies runtime updates automatically by default, so breaking changes to a runtime version could stop your functions from working as expected.
Lambda doesn't provide managed runtimes for language versions which aren't scheduled for LTS release.

Future runtime launch dates

The following list shows the target launch month for upcoming Lambda runtimes. These dates are indicative only and subject to change.

- **.NET 8** - January 2024
- **Ruby 3.3** - March 2024
- **Python 3.13** - November 2024
- **Node.js 22** - November 2024

Runtime deprecation policy

Lambda runtimes for .zip file archives are built around a combination of operating system, programming language, and software libraries that are subject to maintenance and security updates. Lambda's standard deprecation policy is to deprecate a runtime when any major component of the runtime reaches the end of community long-term support (LTS) and security updates are no longer available. Most usually, this is the language runtime, though in some cases, a runtime can be deprecated because the operating system (OS) reaches end of LTS.

After a runtime is deprecated, Lambda no longer applies security patches to that runtime, and functions using that runtime are no longer eligible for technical support.

To learn more about managing runtime upgrades and deprecation, see the following sections and Managing AWS Lambda runtime upgrades on the AWS Compute Blog.

⚠️ Important

Lambda occasionally delays deprecation of a Lambda runtime for a limited period beyond the end of support date of the language version that the runtime supports. During this period, Lambda only applies security patches to the runtime OS. Lambda doesn't apply security patches to programming language runtimes after they reach their end of support date.

Runtime deprecation for Node.js 16
In response to customer feedback, AWS is delaying the deprecation of the Node.js 16 runtime until 6 months after the end of community LTS. The Node.js 16 runtime will be deprecated on the date provided in the Supported Runtimes table. As stated in the preceding note, between the end of LTS on September 11, 2023 and the deprecation date, Lambda will only apply OS patches to the runtime. No security patches for the language runtime will be applied during this period.

Delaying the deprecation of Node.js 16 gives customers using this runtime the opportunity to migrate their functions directly to Node.js 20, skipping Node.js 18.

**Runtime support after deprecation**

For at least 30 days after a runtime is deprecated, you’re still able to create new Lambda functions using that runtime. Starting from 30 days after deprecation, Lambda begins blocking the creation of new functions.

For at least 60 days after a runtime is deprecated, you’re still able to update function code for existing functions. Starting from 60 days after deprecation, Lambda begins blocking the update of function code for existing functions.

Note that the exact length of time for which you can continue to create and update functions isn’t fixed. This period can vary for each deprecation and for different AWS Regions. Nominal dates for the blocking of function creates and updates are provided in the Supported Runtimes table in the first section of this page. Lambda will not start blocking function creates or updates before the dates given in this table.

You can continue to invoke your functions indefinitely after the runtime is deprecated. However, AWS strongly recommends that you migrate functions to a supported runtime so that your functions continue to receive security patches and remain eligible for technical support. You can update a function's configuration to use a newer supported runtime version indefinitely after a runtime is deprecated.

**Receiving runtime deprecation notifications**

When a runtime approaches its deprecation date, Lambda sends you an email alert if any functions in your AWS account use that runtime. Notifications are also displayed in the AWS Health Dashboard and in AWS Trusted Advisor.

- Receiving email notifications:
Lambda sends you an email alert at least **180 days** before a runtime is deprecated. This email lists the $LATEST versions of all functions using the runtime. To see a full list of affected function versions, use Trusted Advisor or see the section called “Listing affected functions using the AWS CLI”.

Lambda sends email notification to your AWS account's primary account contact. For information about viewing or updating the email addresses in your account, see Updating contact information in the AWS General Reference.

- Receiving notifications through the AWS Health Dashboard:

  The AWS Health Dashboard displays a notification at least **180 days** before a runtime is deprecated. Notifications appear on the Your account health page under Other notifications. The Affected resources tab of the notification lists the $LATEST versions of all functions using the runtime.

  ![](note-icon.png)

  **Note**
  To see a full and up-to-date list of affected function versions, use Trusted Advisor or see the section called “Listing affected functions using the AWS CLI”.

  AWS Health Dashboard notifications expire 90 days after the affected runtime is deprecated.

- Using AWS Trusted Advisor

  Trusted Advisor displays a notification **120 days** before a runtime is deprecated. Notifications appear on the Security page. A list of your affected functions is displayed under AWS Lambda Functions Using Deprecated Runtimes. This list of functions shows both $LATEST and published versions and updates automatically to reflect your functions' current status.

  You can turn on weekly email notifications from Trusted Advisor in the Preferences page of the Trusted Advisor console.

**Listing affected functions using the AWS CLI**

In addition to using Trusted Advisor to see a live list of functions affected by scheduled runtime deprecations, you can also use the AWS Command Line Interface (AWS CLI) to list all of your function versions that use a particular runtime. To generate this list, run the following command.
Replace RUNTIME_IDENTIFIER with the name of the runtime that’s being deprecated and choose your own AWS Region. To list only $LATEST function versions, omit --function-version ALL from the command.

```
aws lambda list-functions --function-version ALL --region us-east-1 --output text --query "Functions[?Runtime=='RUNTIME_IDENTIFIER'].FunctionArn"
```

⚠️ Tip

The example command lists functions in the us-east-1 region for a particular AWS account. You’ll need to repeat this command for each region in which your account has functions and for each of your AWS accounts.

You can also use the AWS Config Advanced queries feature to list all your functions that use an affected runtime. This query only returns function $LATEST versions, but you can aggregate queries to list function across all regions and multiple AWS accounts with a single command. To learn more, see [Querying the Current Configuration State of AWS Auto Scaling Resources](https://docs.aws.amazon.com/config/latest/developerguide/config-advanced-queries.html) in the [AWS Config Developer Guide](https://docs.aws.amazon.com/config/latest/developerguide/).

## Deprecated runtimes

The following runtimes have reached end of support:

<table>
<thead>
<tr>
<th>Name</th>
<th>Identifier</th>
<th>Operating system</th>
<th>Deprecation date</th>
<th>Block function create</th>
<th>Block function update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node.js 14</td>
<td>nodejs14.</td>
<td>Amazon Linux 2</td>
<td>Dec 4, 2023</td>
<td>Jan 9, 2024</td>
<td>Feb 8, 2024</td>
</tr>
<tr>
<td>Python 3.7</td>
<td>python3.7</td>
<td>Amazon Linux</td>
<td>Dec 4, 2023</td>
<td>Jan 9, 2024</td>
<td>Feb 8, 2024</td>
</tr>
<tr>
<td>Ruby 2.7</td>
<td>ruby2.7</td>
<td>Amazon Linux 2</td>
<td>Nov 27, 2023</td>
<td>Jan 9, 2024</td>
<td>Feb 8, 2024</td>
</tr>
<tr>
<td>Name</td>
<td>Identifier</td>
<td>Operating system</td>
<td>Deprecation date</td>
<td>Block function create</td>
<td>Block function update</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>------------------</td>
<td>------------------</td>
<td>------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>.NET Core 3.1</td>
<td>dotnetcore3.1</td>
<td>Amazon Linux 2</td>
<td>Apr 3, 2023</td>
<td>Apr 3, 2023</td>
<td>May 3, 2023</td>
</tr>
<tr>
<td>Node.js 12</td>
<td>nodejs12.x</td>
<td>Amazon Linux 2</td>
<td>Mar 31, 2023</td>
<td>Mar 31, 2023</td>
<td>Apr 30, 2023</td>
</tr>
<tr>
<td>Python 3.6</td>
<td>python3.6</td>
<td>Amazon Linux</td>
<td>Jul 18, 2022</td>
<td>Jul 18, 2022</td>
<td>Aug 29, 2022</td>
</tr>
<tr>
<td>.NET 5 (container only)</td>
<td>dotnet5.0</td>
<td>Amazon Linux 2</td>
<td>May 10, 2022</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NET Core 2.1</td>
<td>dotnetcore2.1</td>
<td>Amazon Linux</td>
<td>Jan 5, 2022</td>
<td>Jan 5, 2022</td>
<td>Apr 13, 2022</td>
</tr>
<tr>
<td>Node.js 10</td>
<td>nodejs10.x</td>
<td>Amazon Linux 2</td>
<td>Jul 30, 2021</td>
<td>Jul 30, 2021</td>
<td>Feb 14, 2022</td>
</tr>
<tr>
<td>Ruby 2.5</td>
<td>ruby2.5</td>
<td>Amazon Linux</td>
<td>Jul 30, 2021</td>
<td>Jul 30, 2021</td>
<td>Mar 31, 2022</td>
</tr>
<tr>
<td>Python 2.7</td>
<td>python2.7</td>
<td>Amazon Linux</td>
<td>Jul 15, 2021</td>
<td>Jul 15, 2021</td>
<td>May 30, 2022</td>
</tr>
<tr>
<td>Node.js 8.10</td>
<td>nodejs8.10</td>
<td>Amazon Linux</td>
<td>Mar 6, 2020</td>
<td></td>
<td>Mar 6, 2020</td>
</tr>
<tr>
<td>Node.js 4.3</td>
<td>nodejs4.3</td>
<td>Amazon Linux</td>
<td>Mar 5, 2020</td>
<td></td>
<td>Mar 5, 2020</td>
</tr>
<tr>
<td>Node.js 4.3 edge</td>
<td>nodejs4.3-edge</td>
<td>Amazon Linux</td>
<td>Mar 5, 2020</td>
<td></td>
<td>Apr 30, 2019</td>
</tr>
<tr>
<td>Name</td>
<td>Identifier</td>
<td>Operating system</td>
<td>Deprecation date</td>
<td>Block function create</td>
<td>Block function update</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------</td>
<td>------------------</td>
<td>------------------</td>
<td>-----------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>Node.js 6.10</td>
<td>nodejs6.10</td>
<td>Amazon Linux</td>
<td>Aug 12, 2019</td>
<td>Aug 12, 2019</td>
<td></td>
</tr>
<tr>
<td>.NET Core 1.0</td>
<td>dotnetcore1.0</td>
<td>Amazon Linux</td>
<td>Jun 27, 2019</td>
<td></td>
<td>Jul 30, 2019</td>
</tr>
<tr>
<td>.NET Core 2.0</td>
<td>dotnetcore2.0</td>
<td>Amazon Linux</td>
<td>May 30, 2019</td>
<td></td>
<td>May 30, 2019</td>
</tr>
<tr>
<td>Node.js 0.10</td>
<td>nodejs</td>
<td>Amazon Linux</td>
<td></td>
<td></td>
<td>Oct 31, 2016</td>
</tr>
<tr>
<td>Node.js 6.10</td>
<td>nodejs6.10</td>
<td>Amazon Linux</td>
<td>Aug 12, 2019</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NET 5 (container only)</td>
<td>dotnet5.0</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In almost all cases, the end-of-life date of a language version or operating system is known well in advance. The following links give end-of-life schedules for each language that Lambda supports as a managed runtime.

**Language and framework support policies**

- **Node.js** – [github.com](https://github.com)
- **Python** – [devguide.python.org](https://devguide.python.org)
- **Ruby** – [www.ruby-lang.org](http://www.ruby-lang.org)
- **Java** – [www.oracle.com](http://www.oracle.com) and [Corretto FAQs](https://www.oracle.com/technetwork/)
- **Go** – [golang.org](https://golang.org)
- **.NET** – [dotnet.microsoft.com](https://dotnet.microsoft.com)
Lambda runtime updates

Lambda keeps each managed runtime up to date with security updates, bug fixes, new features, performance enhancements, and support for minor version releases. These runtime updates are published as runtime versions. Lambda applies runtime updates to functions by migrating the function from an earlier runtime version to a new runtime version.

By default, for functions using managed runtimes, Lambda applies runtime updates automatically. With automatic runtime updates, Lambda takes on the operational burden of patching the runtime versions. For most customers, automatic updates are the right choice. For more information, see Runtime management controls.

Lambda also publishes each new runtime version as a container image. To update runtime versions for container-based functions, you must create a new container image from the updated base image and redeploy your function.

Each runtime version is associated with a version number and an ARN (Amazon Resource Name). Runtime version numbers use a numbering scheme that Lambda defines, independent of the version numbers that the programming language uses. The runtime version ARN is a unique identifier for each runtime version.

You can view the ARN of your function's current runtime version in the INIT_START line of your function logs and in the Lambda console.

Runtime versions should not be confused with runtime identifiers. Each runtime has a unique runtime identifier, such as python3.9 or nodejs18.x. These correspond to each major programming language release. Runtime versions describe the patch version of an individual runtime.

Note

The ARN for the same runtime version number can vary between AWS Regions and CPU architectures.

Topics

- Runtime management controls
- Two-phase runtime version rollout
• Roll back a runtime version
• Identifying runtime version changes
• Configure runtime management settings
• Shared responsibility model
• High-compliance applications

Runtime management controls

Lambda strives to provide runtime updates that are backward compatible with existing functions. However, as with software patching, there are rare cases in which a runtime update can negatively impact an existing function. For example, security patches can expose an underlying issue with an existing function that depends on the previous, insecure behavior. Lambda runtime management controls help reduce the risk of impact to your workloads in the rare event of a runtime version incompatibility. For each function version ($LATEST or published version), you can choose one of the following runtime update modes:

• **Auto (default)** – Automatically update to the most recent and secure runtime version using Two-phase runtime version rollout. We recommend this mode for most customers so that you always benefit from runtime updates.

• **Function update** – Update to the most recent and secure runtime version when you update your function. When you update your function, Lambda updates the runtime of your function to the most recent and secure runtime version. This approach synchronizes runtime updates with function deployments, giving you control over when Lambda applies runtime updates. With this mode, you can detect and mitigate rare runtime update incompatibilities early. When using this mode, you must regularly update your functions to keep their runtime up to date.

• **Manual** – Manually update your runtime version. You specify a runtime version in your function configuration. The function uses this runtime version indefinitely. In the rare case in which a new runtime version is incompatible with an existing function, you can use this mode to roll back your function to an earlier runtime version. We recommend against using Manual mode to try to achieve runtime consistency across deployments. For more information, see Roll back a runtime version.

Responsibility for applying runtime updates to your functions varies according to which runtime update mode you choose. For more information, see Shared responsibility model.
Two-phase runtime version rollout

Lambda introduces new runtime versions in the following order:

1. In the first phase, Lambda applies the new runtime version whenever you create or update a function. A function gets updated when you call the `UpdateFunctionCode` or `UpdateFunctionConfiguration` API operations.

2. In the second phase, Lambda updates any function that uses the **Auto** runtime update mode and that hasn't already been updated to the new runtime version.

The overall duration of the rollout process varies according to multiple factors, including the severity of any security patches included in the runtime update.

If you're actively developing and deploying your functions, you will most likely pick up new runtime versions during the first phase. This synchronizes runtime updates with function updates. In the rare event that the latest runtime version negatively impacts your application, this approach lets you take prompt corrective action. Functions that aren't in active development still receive the operational benefit of automatic runtime updates during the second phase.

This approach doesn't affect functions set to **Function update** or **Manual** mode. Functions using **Function update** mode receive the latest runtime updates only when you create or update them. Functions using **Manual** mode don't receive runtime updates.

Lambda publishes new runtime versions in a gradual, rolling fashion across AWS Regions. If your functions are set to **Auto** or **Function update** modes, it's possible that functions deployed at the same time to different Regions, or at different times in the same Region, will pick up different runtime versions. Customers who require guaranteed runtime version consistency across their environments should use container images to deploy their Lambda functions. The **Manual** mode is designed as a temporary mitigation to enable runtime version rollback in the rare event that a runtime version is incompatible with your function.

Roll back a runtime version

In the rare event that a new runtime version is incompatible with your existing function, you can roll back its runtime version to an earlier one. This keeps your application working and minimizes disruption, providing time to remedy the incompatibility before returning to the latest runtime version.
Lambda doesn't impose a time limit on how long you can use any particular runtime version. However, we strongly recommend updating to the latest runtime version as soon as possible to benefit from the latest security patches, performance improvements, and features. Lambda provides the option to roll back to an earlier runtime version only as a temporary mitigation in the rare event of a runtime update compatibility issue. Functions using an earlier runtime version for an extended period may eventually experience degraded performance or issues, such as a certificate expiry, which can cause them to stop working properly.

You can roll back a runtime version in the following ways:

- **Using the Manual runtime update mode**
- **Using published function versions**

For more information, see [Introducing AWS Lambda runtime management controls](https://aws.amazon.com/language/compute/blog/introducing-aws-lambda-runtime-management-controls/) on the AWS Compute Blog.

### Roll back a runtime version using Manual runtime update mode

If you're using the **Auto** runtime version update mode, or you're using the $LATEST runtime version, you can roll back your runtime version using the **Manual** mode. For the function version you want to roll back, change the runtime version update mode to **Manual** and specify the ARN of the previous runtime version. For more information about finding the ARN of the previous runtime version, see [Identifying runtime version changes](https).

#### Note

If the $LATEST version of your function is configured to use **Manual** mode, then you can't change the CPU architecture or runtime version that your function uses. To make these changes, you must change to **Auto** or **Function update** mode.

### Roll back a runtime version using published function versions

Published function versions are an immutable snapshot of the $LATEST function code and configuration at the time that you created them. In **Auto** mode, Lambda automatically updates the runtime version of published function versions during phase two of the runtime version rollout. In **Function update** mode, Lambda doesn't update the runtime version of published function versions.
Published function versions using **Function update** mode therefore create a static snapshot of the function code, configuration, and runtime version. By using **Function update** mode with function versions, you can synchronize runtime updates with your deployments. You can also coordinate rollback of code, configuration, and runtime versions by redirecting traffic to an earlier published function version. You can integrate this approach into your continuous integration and continuous delivery (CI/CD) for fully automatic rollback in the rare event of runtime update incompatibility. When using this approach, you must update your function regularly and publish new function versions to pick up the latest runtime updates. For more information, see [Shared responsibility model](#).

### Identifying runtime version changes

The runtime version number and ARN are logged in the **INIT_START** log line, which Lambda emits to CloudWatch Logs each time that it creates a new execution environment. Because the execution environment uses the same runtime version for all function invocations, Lambda emits the **INIT_START** log line only when Lambda executes the init phase. Lambda doesn't emit this log line for each function invocation. Lambda emits the log line to CloudWatch Logs, but it is not visible in the console.

**Example Example INIT_START log line**

```
INIT_START Runtime Version: python:3.9.v14    Runtime Version ARN: arn:aws:lambda:eu-south-1::runtime:7b620fc2e66107a1046b140b9d320295811af3ad5d4c6a011fad1fa65127e9e61
```

Rather than working directly with the logs, you can use [Amazon CloudWatch Contributor Insights](#) to identify transitions between runtime versions. The following rule counts the distinct runtime versions from each **INIT_START** log line. To use the rule, replace the example log group name `/aws/lambda/*` with the appropriate prefix for your function or group of functions.

```json
{
    "Schema": {
        "Name": "CloudWatchLogRule",
        "Version": 1
    },
    "AggregateOn": "Count",
    "Contribution": {
        "Filters": [
            {
                "Match": "eventType",
                "Values": "]
            }
        ]
    }
}
```
The following CloudWatch Contributor Insights report shows an example of a runtime version transition as captured by the preceding rule. The orange line shows execution environment initialization for the earlier runtime version (python:3.9.v12), and the blue line shows execution environment initialization for the new runtime version (python:3.9.v14).
Configure runtime management settings

You can configure runtime management settings using the Lambda console or the AWS Command Line Interface (AWS CLI).

ℹ️ Note
You can configure runtime management settings separately for each function version.

To configure how Lambda updates your runtime version (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. On the Code tab, under Runtime settings, choose Edit runtime management configuration.
4. Under Runtime management configuration, choose one of the following:
   - To have your function update to the latest runtime version automatically, choose Auto.
   - To have your function update to the latest runtime version when you change the function, choose Function update.
   - To have your function update to the latest runtime version only when you change the runtime version ARN, choose Manual.

ℹ️ Note
You can find the runtime version ARN under Runtime management configuration. You can also find the ARN in the INIT_START line of your function logs.

5. Choose Save.

To configure how Lambda updates your runtime version (AWS CLI)

To configure runtime management for a function, you can use the `put-runtime-management-config` AWS CLI command, together with the runtime update mode. When using Manual mode, you must also provide the runtime version ARN.

```
```
You should see output similar to the following:

```json
{
    "UpdateRuntimeOn": "Manual",
    "FunctionArn": "arn:aws:lambda:eu-west-1:069549076217:function:myfunction",
    "RuntimeVersionArn": "arn:aws:lambda:eu-west-1::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1"
}
```

## Shared responsibility model

Lambda is responsible for curating and publishing security updates for all supported managed runtimes and container images. Responsibility for updating existing functions to use the latest runtime version varies depending on which runtime update mode you use.

Lambda is responsible for applying runtime updates to all functions configured to use the **Auto** runtime update mode.

For functions configured with the **Function update** runtime update mode, you're responsible for regularly updating your function. Lambda is responsible for applying runtime updates when you make those updates. If you don't update your function, then Lambda doesn't update the runtime. If you don't regularly update your function, then we strongly recommend configuring it for automatic runtime updates so that it continues to receive security updates.

For functions configured to use the **Manual** runtime update mode, you're responsible for updating your function to use the latest runtime version. We strongly recommend that you use this mode only to roll back the runtime version as a temporary mitigation in the rare event of runtime update incompatibility. We also recommend that you change to **Auto** mode as quickly as possible to minimize the time in which your functions aren't patched.

If you're using container images to deploy your functions, then Lambda is responsible for publishing updated base images. In this case, you're responsible for rebuilding your function's container image from the latest base image and redeploying the container image.

This is summarized in the following table:
<table>
<thead>
<tr>
<th>Deployment mode</th>
<th>Lambda's responsibility</th>
<th>Customer's responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Managed runtime, Auto mode</td>
<td>Publish new runtime versions containing the latest patches. Apply runtime patches to existing functions.</td>
<td>Roll back to a previous runtime version in the rare event of a runtime update compatibility issue.</td>
</tr>
<tr>
<td>Managed runtime, Function update mode</td>
<td>Publish new runtime versions containing the latest patches.</td>
<td>Update functions regularly to pick up the latest runtime version. Switch a function to Auto mode when you’re not regularly updating the function. Roll back to a previous runtime version in the rare event of a runtime update compatibility issue.</td>
</tr>
<tr>
<td>Managed runtime, Manual mode</td>
<td>Publish new runtime versions containing the latest patches.</td>
<td>Use this mode only for temporary runtime rollback in the rare event of a runtime update compatibility issue. Switch functions to Auto or Function update mode and the latest runtime version as soon as possible.</td>
</tr>
<tr>
<td>Container image</td>
<td>Publish new container images containing the latest patches.</td>
<td>Redeploy functions regularly using the latest container base image to pick up the latest patches.</td>
</tr>
</tbody>
</table>

For more information about shared responsibility with AWS, see [Shared Responsibility Model](https://aws.amazon.com/aws-cloud-security/model) on the AWS Cloud Security site.

**High-compliance applications**

To meet patching requirements, Lambda customers typically rely on automatic runtime updates. If your application is subject to strict patching freshness requirements, you may want to limit use
of earlier runtime versions. You can restrict Lambda's runtime management controls by using AWS Identity and Access Management (IAM) to deny users in your AWS account access to the `PutRuntimeManagementConfig` API operation. This operation is used to choose the runtime update mode for a function. Denying access to this operation causes all functions to default to the **Auto** mode. You can apply this restriction across your organization by using a service control policies (SCP). In the event that you must roll back a function to an earlier runtime version, you can grant a policy exception on a case-by-case basis.
Modifying the runtime environment

You can use internal extensions to modify the runtime process. Internal extensions are not separate processes—they run as part of the runtime process.

Lambda provides language-specific environment variables that you can set to add options and tools to the runtime. Lambda also provides wrapper scripts, which allow Lambda to delegate the runtime startup to your script. You can create a wrapper script to customize the runtime startup behavior.

Language-specific environment variables

Lambda supports configuration-only ways to enable code to be pre-loaded during function initialization through the following language-specific environment variables:

- **JAVA_TOOL_OPTIONS** – On Java, Lambda supports this environment variable to set additional command-line variables in Lambda. This environment variable allows you to specify the initialization of tools, specifically the launching of native or Java programming language agents using the agentlib or javaagent options. For more information, see [JAVA_TOOL_OPTIONS environment variable](#).
- **NODE_OPTIONS** – Available in Node.js runtimes.
- **DOTNET_STARTUP_HOOKS** – On .NET Core 3.1 and above, this environment variable specifies a path to an assembly (dll) that Lambda can use.

Using language-specific environment variables is the preferred way to set startup properties.

Wrapper scripts

You can create a wrapper script to customize the runtime startup behavior of your Lambda function. A wrapper script enables you to set configuration parameters that cannot be set through language-specific environment variables.

⚠️ Note

Invocations may fail if the wrapper script does not successfully start the runtime process.
Wrapper scripts are supported on all native Lambda runtimes which use the Amazon Linux 2 operating system. The custom provided al2 runtime does not support wrapper scripts.

When you use a wrapper script for your function, Lambda starts the runtime using your script. Lambda sends to your script the path to the interpreter and all of the original arguments for the standard runtime startup. Your script can extend or transform the startup behavior of the program. For example, the script can inject and alter arguments, set environment variables, or capture metrics, errors, and other diagnostic information.

You specify the script by setting the value of the AWS_LAMBDA_EXEC_WRAPPER environment variable as the file system path of an executable binary or script.

**Example: Create and use a wrapper script with Python 3.8**

In the following example, you create a wrapper script to start the Python interpreter with the -X importtime option. When you run the function, Lambda generates a log entry to show the duration of the import time for each import.

**To create and use a wrapper script with Python 3.8**

1. To create the wrapper script, paste the following code into a file named `importtime_wrapper`:

   ```bash
   #!/bin/bash

   # the path to the interpreter and all of the originally intended arguments
   args="@"

   # the extra options to pass to the interpreter
   extra_args="-X importtime"

   # insert the extra options
   args="${args[@]:0:$#-1} ${extra_args[@]} ${args[@]: -1}"

   # start the runtime with the extra options
   exec "${args[@]}"
   
   
   2. To give the script executable permissions, enter `chmod +x importtime_wrapper` from the command line.
3. Deploy the script as a **Lambda layer**.

4. Create a function using the Lambda console.
   a. Open the **Lambda console**.
   b. Choose **Create function**.
   c. Under **Basic information**, for **Function name**, enter *wrapper-test-function*.
   d. For **Runtime**, choose **Python 3.8**.
   e. Choose **Create function**.

5. Add the layer to your function.
   a. Choose your function, and then choose **Code** if it is not already selected.
   b. Choose **Add a layer**.
   c. Under **Choose a layer**, choose the **Name** and **Version** of the compatible layer that you created earlier.
   d. Choose **Add**.

6. Add the code and the environment variable to your function.
   a. In the function **code editor**, paste the following function code:
      
      ```python
      import json

      def lambda_handler(event, context):
          # TODO implement
          return {
              'statusCode': 200,
              'body': json.dumps('Hello from Lambda!')
          }
      
      # For Key, enter AWS_LAMBDA_EXEC_WRAPPER.
      # For Value, enter /opt/importtime_wrapper.
      # Choose Save.
      # Choose Test.
      ```
   b. Choose **Save**.
   c. Under **Environment variables**, choose **Edit**.
   d. Choose **Add environment variable**.
   e. For **Key**, enter **AWS_LAMBDA_EXEC_WRAPPER**.
   f. For **Value**, enter **/opt/importtime_wrapper**.
   g. Choose **Save**.
   h. To run the function, choose **Test**.
Because your wrapper script started the Python interpreter with the `-X importtime` option, the logs show the time required for each import. For example:

```plaintext
... 
2020-06-30T18:48:46.780+01:00 import time: 213 | 213 | simplejson 
2020-06-30T18:48:46.780+01:00 import time: 50  | 263  | simplejson.raw_json 
... 
```
Lambda runtime API

AWS Lambda provides an HTTP API for custom runtimes to receive invocation events from Lambda and send response data back within the Lambda execution environment.

The OpenAPI specification for the runtime API version **2018-06-01** is available in runtime-api.zip

To create an API request URL, runtimes get the API endpoint from the AWS_LAMBDA_RUNTIME_API environment variable, add the API version, and add the desired resource path.

**Example Request**

```
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next"
```

**API methods**

- Next invocation
- Invocation response
- Initialization error
- Invocation error

**Next invocation**

**Path** – /runtime/invocation/next

**Method** – GET
The runtime sends this message to Lambda to request an invocation event. The response body contains the payload from the invocation, which is a JSON document that contains event data from the function trigger. The response headers contain additional data about the invocation.

**Response headers**

- **Lambda-Runtime-Aws-Request-Id** – The request ID, which identifies the request that triggered the function invocation.
  
  For example, 8476a536-e9f4-11e8-9739-2dfe598c3fcd.

- **Lambda-Runtime-Deadline-Ms** – The date that the function times out in Unix time milliseconds.
  
  For example, 1542409706888.

- **Lambda-Runtime-Invoked-Function-Arn** – The ARN of the Lambda function, version, or alias that's specified in the invocation.
  
  For example, arn:aws:lambda:us-east-2:123456789012:function:custom-runtime.

- **Lambda-Runtime-Trace-Id** – The [AWS X-Ray tracing header](https://docs.aws.amazon.com/xray/latest/dg/what-is.html).
  
  For example, Root=1-5bef4de7-ad49b0e87f6ef6c87fc2e700;Parent=9a9197af755a6419;Sampled=1.

- **Lambda-Runtime-Client-Context** – For invocations from the AWS Mobile SDK, data about the client application and device.

- **Lambda-Runtime-Cognito-Identity** – For invocations from the AWS Mobile SDK, data about the Amazon Cognito identity provider.

Do not set a timeout on the GET request as the response may be delayed. Between when Lambda bootstraps the runtime and when the runtime has an event to return, the runtime process may be frozen for several seconds.

The request ID tracks the invocation within Lambda. Use it to specify the invocation when you send the response.

The tracing header contains the trace ID, parent ID, and sampling decision. If the request is sampled, the request was sampled by Lambda or an upstream service. The runtime should set the _X_AMZN_TRACE_ID with the value of the header. The X-Ray SDK reads this to get the IDs and determine whether to trace the request.
Invocation response

Path – /runtime/invocation/{AwsRequestId}/response

Method – POST

After the function has run to completion, the runtime sends an invocation response to Lambda. For synchronous invocations, Lambda sends the response to the client.

Example success request

```bash
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/${REQUEST_ID}/response" -d "SUCCESS"
```

Initialization error

If the function returns an error or the runtime encounters an error during initialization, the runtime uses this method to report the error to Lambda.

Path – /runtime/init/error

Method – POST

Headers

Lambda-Runtime-Function-Error-Type – Error type that the runtime encountered. Required: no.

This header consists of a string value. Lambda accepts any string, but we recommend a format of <category.reason>. For example:

- Runtime.NoSuchHandler
- Runtime.APIKeyNotFound
- Runtime.ConfigInvalid
- Runtime.UnknownReason

Body parameters
ErrorRequest – Information about the error. Required: no.

This field is a JSON object with the following structure:

```json
{
    errorMessage: string (text description of the error),
    errorType: string,
    stackTrace: array of strings
}
```

Note that Lambda accepts any value for errorType.

The following example shows a Lambda function error message in which the function could not parse the event data provided in the invocation.

**Example Function error**

```json
{
    "errorMessage" : "Error parsing event data.",
    "errorType" : "InvalidEventDataException",
    "stackTrace": []
}
```

**Response body parameters**

- **StatusResponse** – String. Status information, sent with 202 response codes.
- **ErrorResponse** – Additional error information, sent with the error response codes. ErrorResponse contains an error type and an error message.

**Response codes**

- 202 – Accepted
- 403 – Forbidden
- 500 – Container error. Non-recoverable state. Runtime should exit promptly.

**Example initialization error request**

```plaintext
ERROR="{"errorMessage" : "Failed to load function.\", "errorType" : "InvalidFunctionException\"}"
```

Initialization error
Invocation error

If the function returns an error or the runtime encounters an error, the runtime uses this method to report the error to Lambda.

Path – /runtime/invocation/AwsRequestId/error

Method – POST

Headers

Lambda-Runtime-Function-Error-Type – Error type that the runtime encountered. Required: no.

This header consists of a string value. Lambda accepts any string, but we recommend a format of <category.reason>. For example:

- Runtime.NoSuchHandler
- Runtime.APIKeyNotFound
- Runtime.ConfigInvalid
- Runtime.UnknownReason

Body parameters

ErrorRequest – Information about the error. Required: no.

This field is a JSON object with the following structure:

```json
{
  errorMessage: string (text description of the error),
  errorType: string,
  stackTrace: array of strings
}
```

Note that Lambda accepts any value for errorType.

The following example shows a Lambda function error message in which the function could not parse the event data provided in the invocation.
Example Function error

```
{
    "errorMessage" : "Error parsing event data.",
    "errorType" : "InvalidEventDataException",
    "StackTrace": []
}
```

Response body parameters

- **StatusResponse** – String. Status information, sent with 202 response codes.
- **ErrorResponse** – Additional error information, sent with the error response codes. ErrorResponse contains an error type and an error message.

Response codes

- 202 – Accepted
- 400 – Bad Request
- 403 – Forbidden
- 500 – Container error. Non-recoverable state. Runtime should exit promptly.

Example error request

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
ERROR="\"errorMessage\" : \"Error parsing event data.\", \"errorType\" : \"InvalidEventDataException\"
```
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/error"
  -d "$ERROR" --header "Lambda-Runtime-Function-Error-Type: Unhandled"
OS-only runtimes for AWS Lambda

Lambda provides managed runtimes for Java, Python, Node.js, .NET, and Ruby. To create Lambda functions in a programming language that is not available as a managed runtime, use an OS-only runtime (the provided runtime family). There are three primary use cases for OS-only runtimes:

- **Native ahead-of-time (AOT) compilation**: Languages such as Go, Rust, and C++ compile natively to an executable binary, which doesn't require a dedicated language runtime. These languages only need an OS environment in which the compiled binary can run. You can also use Lambda OS-only runtimes to deploy binaries compiled with .NET Native AOT and Java GraalVM Native.

  You must include a runtime interface client in your binary. The runtime interface client calls the Lambda runtime API to retrieve function invocations and then calls your function handler. Lambda provides runtime interface clients for Go, .NET Native AOT, C++, and Rust (experimental).

  You must compile your binary for a Linux environment and for the same instruction set architecture that you plan to use for the function (x86_64 or arm64).

- **Third-party runtimes**: You can run Lambda functions using off-the-shelf runtimes such as Bref for PHP or the Swift AWS Lambda Runtime for Swift.

- **Custom runtimes**: You can build your own runtime for a language or language version that Lambda doesn't provide a managed runtime for, such as Node.js 19. For more information, see Building a custom runtime for AWS Lambda. This is the least common use case for OS-only runtimes.

Lambda supports the following OS-only runtimes:

### OS-only

<table>
<thead>
<tr>
<th>Name</th>
<th>Identifier</th>
<th>Operating system</th>
<th>Deprecation date</th>
<th>Block function create</th>
<th>Block function update</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS-only Runtime provided.al2023</td>
<td>Amazon Linux 2023</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS-only Runtime provided.al2</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Identifier</td>
<td>Operating system</td>
<td>Deprecation date</td>
<td>Block function create</td>
<td>Block function update</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------</td>
<td>-----------------------</td>
<td>-------------------</td>
<td>-----------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>OS-only Runtime</td>
<td>provided</td>
<td>Amazon Linux</td>
<td>Jan 8, 2024</td>
<td>Feb 8, 2024</td>
<td>Mar 12, 2024</td>
</tr>
</tbody>
</table>

The Amazon Linux 2023 (provided.al2023) runtime provides several advantages over Amazon Linux 2, including a smaller deployment footprint and updated versions of libraries such as glibc.

The provided.al2023 runtime uses dnf as the package manager instead of yum, which is the default package manager in Amazon Linux 2. For more information about the differences between provided.al2023 and provided.al2, see [Introducing the Amazon Linux 2023 runtime for AWS Lambda](https://aws.amazon.com/blogs/compute/introducing-the-amazon-linux-2023-runtime-for-aws-lambda) on the AWS Compute Blog.
Building a custom runtime for AWS Lambda

You can implement an AWS Lambda runtime in any programming language. A runtime is a program that runs a Lambda function's handler method when the function is invoked. You can include the runtime in your function's deployment package or distribute it in a layer. When you create the Lambda function, choose an OS-only runtime (the provided runtime family).

Note

Creating a custom runtime is an advanced use case. If you're looking for information about compiling to a native binary or using a third-party off-the-shelf runtime, see OS-only runtimes for AWS Lambda.

For a walkthrough of the custom runtime deployment process, see Tutorial: Building a custom runtime. You can also explore a custom runtime implemented in C++ at awslabs/aws-lambda-cpp on GitHub.

Topics

• Requirements
• Implementing response streaming in a custom runtime

Requirements

Custom runtimes must complete certain initialization and processing tasks. A runtime runs the function's setup code, reads the handler name from an environment variable, and reads invocation events from the Lambda runtime API. The runtime passes the event data to the function handler, and posts the response from the handler back to Lambda.

Initialization tasks

The initialization tasks run once per instance of the function to prepare the environment to handle invocations.

• Retrieve settings – Read environment variables to get details about the function and environment.
• **_HANDLER** – The location to the handler, from the function's configuration. The standard format is *file.method*, where file is the name of the file without an extension, and method is the name of a method or function that's defined in the file.

• **LAMBDA_TASK_ROOT** – The directory that contains the function code.

• **AWS_LAMBDA_RUNTIME_API** – The host and port of the runtime API.

For a full list of available variables, see [Defined runtime environment variables](#).

• **Initialize the function** – Load the handler file and run any global or static code that it contains. Functions should create static resources like SDK clients and database connections once, and reuse them for multiple invocations.

• **Handle errors** – If an error occurs, call the [initialization error](#) API and exit immediately.

Initialization counts towards billed execution time and timeout. When an execution triggers the initialization of a new instance of your function, you can see the initialization time in the logs and [AWS X-Ray trace](#).

**Example log**

```
REPORT RequestId: f8ac1208... Init Duration: 48.26 ms   Duration: 237.17 ms   Billed Duration: 300 ms   Memory Size: 128 MB   Max Memory Used: 26 MB
```

**Processing tasks**

While it runs, a runtime uses the [Lambda runtime interface](#) to manage incoming events and report errors. After completing initialization tasks, the runtime processes incoming events in a loop. In your runtime code, perform the following steps in order.

• **Get an event** – Call the [next invocation](#) API to get the next event. The response body contains the event data. Response headers contain the request ID and other information.

• **Propagate the tracing header** – Get the X-Ray tracing header from the Lambda-Runtime-Trace-Id header in the API response. Set the _X_AMZN_TRACE_ID environment variable locally with the same value. The X-Ray SDK uses this value to connect trace data between services.

• **Create a context object** – Create an object with context information from environment variables and headers in the API response.

• **Invoke the function handler** – Pass the event and context object to the handler.

• **Handle the response** – Call the [invocation response](#) API to post the response from the handler.
- **Handle errors** – If an error occurs, call the [invocation error](#) API.

- **Cleanup** – Release unused resources, send data to other services, or perform additional tasks before getting the next event.

**Entrypoint**

A custom runtime's entry point is an executable file named `bootstrap`. The bootstrap file can be the runtime, or it can invoke another file that creates the runtime. If the root of your deployment package doesn't contain a file named `bootstrap`, Lambda looks for the file in the function's layers. If the `bootstrap` file doesn't exist or isn't executable, your function returns a `Runtime.InvalidEntrypoint` error upon invocation.

Here's an example `bootstrap` file that uses a bundled version of Node.js to run a JavaScript runtime in a separate file named `runtime.js`.

**Example bootstrap**

```bash
#!/bin/sh
  cd $LAMBDA_TASK_ROOT
  ./node-v11.1.0-linux-x64/bin/node runtime.js
```

**Implementing response streaming in a custom runtime**

For **response streaming functions**, the response and error endpoints have slightly modified behavior that lets the runtime stream partial responses to the client and return payloads in chunks. For more information about the specific behavior, see the following:

- `/runtime/invocation/AwsRequestId/response` – Propagates the Content-Type header from the runtime to send to the client. Lambda returns the response payload in chunks via HTTP/1.1 chunked transfer encoding. The response stream can be a maximum size of 20 MiB. To stream the response to Lambda, the runtime must:
  - Set the `Lambda-Runtime-Function-Response-Mode` HTTP header to streaming.
  - Set the `Transfer-Encoding` header to chunked.
  - Write the response conforming to the HTTP/1.1 chunked transfer encoding specification.
  - Close the underlying connection after it has successfully written the response.
• /runtime/invocation/AwsRequestId/error – The runtime can use this endpoint to report function or runtime errors to Lambda, which also accepts the Transfer-Encoding header. This endpoint can only be called before the runtime begins sending an invocation response.

• Report midstream errors using error trailers in /runtime/invocation/AwsRequestId/response – To report errors that occur after the runtime starts writing the invocation response, the runtime can optionally attach HTTP trailing headers named Lambda-Runtime-Function-Error-Type and Lambda-Runtime-Function-Error-Body. Lambda treats this as a successful response and forwards the error metadata that the runtime provides to the client.

   Note

   To attach trailing headers, the runtime must set the Trailer header value at the beginning of the HTTP request. This is a requirement of the HTTP/1.1 chunked transfer encoding specification.

• Lambda-Runtime-Function-Error-Type – The error type that the runtime encountered. This header consists of a string value. Lambda accepts any string, but we recommend a format of <category.reason>. For example, Runtime.APIKeyNotFound.

• Lambda-Runtime-Function-Error-Body – Base64-encoded information about the error.
Tutorial: Building a custom runtime

In this tutorial, you create a Lambda function with a custom runtime. You start by including the runtime in the function's deployment package. Then you migrate it to a layer that you manage independently from the function. Finally, you share the runtime layer with the world by updating its resource-based permissions policy.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven’t already, follow the instructions in Create a Lambda function with the console to create your first Lambda function.

To complete the following steps, you need the AWS Command Line Interface (AWS CLI) version 2. Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux. Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

You need an IAM role to create a Lambda function. The role needs permission to send logs to CloudWatch Logs and access the AWS services that your function uses. If you don't have a role for function development, create one now.
To create an execution role

1. Open the roles page in the IAM console.
2. Choose Create role.
3. Create a role with the following properties.

   - Trusted entity – Lambda.
   - Permissions – AWSLambdaBasicExecutionRole.
   - Role name – lambda-role.

   The AWSLambdaBasicExecutionRole policy has the permissions that the function needs to write logs to CloudWatch Logs.

Create a function

Create a Lambda function with a custom runtime. This example includes two files: a runtime bootstrap file and a function handler. Both are implemented in Bash.

1. Create a directory for the project, and then switch to that directory.

   ```sh
   mkdir runtime-tutorial
   cd runtime-tutorial
   ```

2. Create a new file called bootstrap. This is the custom runtime.

   Example bootstrap

   ```sh
   #!/bin/sh
   set -euo pipefail

   # Initialization - load function handler
   source $LAMBDA_TASK_ROOT"$(echo $_HANDLER | cut -d -f1).sh"

   # Processing
   while true
   do
     HEADERS="$(mktemp)"
     # Get an event. The HTTP request will block until one is received
   ```
EVENT_DATA=$(curl -s -LD "$HEADERS" "http://
${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")

# Extract request ID by scraping response headers received above
REQUEST_ID=$(grep -F 'Lambda-Runtime-Aws-Request-Id' "$HEADERS" | tr -d '[:space:]' | cut -d: -f2)

# Run the handler function from the script
RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

# Send the response
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "$RESPONSE"
done

The runtime loads a function script from the deployment package. It uses two variables to locate the script. LAMBDA_TASK_ROOT tells it where the package was extracted, and _HANDLER includes the name of the script.

After the runtime loads the function script, it uses the runtime API to retrieve an invocation event from Lambda, passes the event to the handler, and posts the response back to Lambda. To get the request ID, the runtime saves the headers from the API response to a temporary file, and reads the Lambda-Runtime-Aws-Request-Id header from the file.

---

**Note**

Runtimes have additional responsibilities, including error handling, and providing context information to the handler. For details, see [Requirements](#).

---

3. Create a script for the function. The following example script defines a handler function that takes event data, logs it to stderr, and returns it.

**Example function.sh**

```bash
function handler () {
    EVENT_DATA=$1
    echo "EVENT_DATA" 1>&2;
    RESPONSE="Echoing request: '$EVENT_DATA'"
    echo $RESPONSE
}
```
The `runtime-tutorial` directory should now look like this:

```bash
runtime-tutorial
# bootstrap
# function.sh
```

4. Make the files executable and add them to a `.zip` file archive. This is the deployment package.

```bash
chmod 755 function.sh bootstrap
zip function.zip function.sh bootstrap
```

5. Create a function named `bash-runtime`. For `--role`, enter the ARN of your Lambda execution role.

```bash
aws lambda create-function --function-name bash-runtime \
--zip-file fileb://function.zip --handler function.handler --runtime provided.al2023 \
--role arn:aws:iam::123456789012:role/lambda-role
```

6. Invoke the function.

```bash
aws lambda invoke --function-name bash-runtime --payload ':{"text":"Hello"}' response.txt --cli-binary-format raw-in-base64-out
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/guide/setting-cli.html) in the [AWS Command Line Interface User Guide for Version 2](https://docs.aws.amazon.com/cli/latest/userguide/). You should see a response like this:

```json
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

7. Verify the response.
Create a layer

To separate the runtime code from the function code, create a layer that only contains the runtime. Layers let you develop your function's dependencies independently, and can reduce storage usage when you use the same layer with multiple functions. For more information, see Working with Lambda layers.

1. Create a .zip file that contains the `bootstrap` file.

   ```bash
   zip runtime.zip bootstrap
   ```

2. Create a layer with the `publish-layer-version` command.

   ```bash
   aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://runtime.zip
   ```

   This creates the first version of the layer.

Update the function

To use the runtime layer in the function, configure the function to use the layer, and remove the runtime code from the function.

1. Update the function configuration to pull in the layer.

   ```bash
   aws lambda update-function-configuration --function-name bash-runtime \
   --layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:1
   ```
This adds the runtime to the function in the /opt directory. To ensure that Lambda uses the runtime in the layer, you must remove the bootstrap from the function's deployment package, as shown in the next two steps.

2. Create a .zip file that contains the function code.

    ```bash
    zip function-only.zip function.sh
    ```

3. Update the function code to only include the handler script.

    ```bash
    aws lambda update-function-code --function-name bash-runtime --zip-file fileb://function-only.zip
    ```

4. Invoke the function to confirm that it works with the runtime layer.

    ```bash
    aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}' response.txt --cli-binary-format raw-in-base64-out
    ```

The cli-binary-format option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the AWS Command Line Interface User Guide for Version 2.

You should see a response like this:

```json
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

5. Verify the response.

    ```bash
    cat response.txt
    ```

You should see a response like this:

```
Echoing request: '{"text":"Hello"}'
```
Update the runtime

1. To log information about the execution environment, update the runtime script to output environment variables.

   **Example bootstrap**

   ```sh
   #!/bin/sh

   set -euo pipefail

   # Configure runtime to output environment variables
   echo "## Environment variables:"
   env

   # Load function handler
   source $LAMBDA_TASK_ROOT/$(echo $_HANDLER | cut -d. -f1).sh

   # Processing
   while true
   do
     HEADERS="$(mktemp)"
     # Get an event. The HTTP request will block until one is received
     EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
     ${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")

     # Extract request ID by scraping response headers received above
     REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
     '[:space:]' | cut -d: -f2)

     # Run the handler function from the script
     RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

     # Send the response
     curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/REQUEST_ID/
     response" -d "$RESPONSE"
   done
   ```

2. Create a .zip file that contains the new version of the bootstrap file.

   ```sh
   zip runtime.zip bootstrap
   ```

3. Create a new version of the bash-runtime layer.
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://runtime.zip

4. Configure the function to use the new version of the layer.

```bash
aws lambda update-function-configuration --function-name bash-runtime \
```

**Share the layer**

To grant layer-usage permission to another account, add a statement to the layer version's permissions policy using the `add-layer-version-permission` command. In each statement, you can grant permission to a single account, all accounts, or an organization.

The following example grants account 111122223333 access to version 2 of the bash-runtime layer.

```bash
aws lambda add-layer-version-permission --layer-name bash-runtime --statement-id xaccount \n   --action lambda:GetLayerVersion --principal 111122223333 --version-number 2 --output text
```

You should see output similar to the following:

```
e210ffdc-e901-43b0-824b-5fcd0dd26d16    {"Sid":"xaccount","Effect":"Allow","Principal":
   {"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
   east-1:123456789012:layer:bash-runtime:2"}
```

Permissions apply only to a single layer version. Repeat the process each time that you create a new layer version.

**Clean up**

Delete each version of the layer.

```bash
aws lambda delete-layer-version --layer-name bash-runtime --version-number 1
aws lambda delete-layer-version --layer-name bash-runtime --version-number 2
```
Because the function holds a reference to version 2 of the layer, it still exists in Lambda. The function continues to work, but functions can no longer be configured to use the deleted version. If you modify the list of layers on the function, you must specify a new version or omit the deleted layer.

Delete the function with the `delete-function` command.

```
aws lambda delete-function --function-name bash-runtime
```
Using AVX2 vectorization in Lambda

Advanced Vector Extensions 2 (AVX2) is a vectorization extension to the Intel x86 instruction set that can perform single instruction multiple data (SIMD) instructions over vectors of 256 bits. For vectorizable algorithms with highly parallelizable operation, using AVX2 can enhance CPU performance, resulting in lower latencies and higher throughput. Use the AVX2 instruction set for compute-intensive workloads such as machine learning inferencing, multimedia processing, scientific simulations, and financial modeling applications.

Note
Lambda arm64 uses NEON SIMD architecture and does not support the x86 AVX2 extensions.

To use AVX2 with your Lambda function, make sure that your function code is accessing AVX2-optimized code. For some languages, you can install the AVX2-supported version of libraries and packages. For other languages, you can recompile your code and dependencies with the appropriate compiler flags set (if the compiler supports auto-vectorization). You can also compile your code with third-party libraries that use AVX2 to optimize math operations. For example, Intel Math Kernel Library (Intel MKL), OpenBLAS (Basic Linear Algebra Subprograms), and AMD BLAS-like Library Instantiation Software (BLIS). Auto-vectorized languages, such as Java, automatically use AVX2 for computations.

You can create new Lambda workloads or move existing AVX2-enabled workloads to Lambda at no additional cost.

For more information about AVX2, see Advanced Vector Extensions 2 in Wikipedia.

Compiling from source

If your Lambda function uses a C or C++ library to perform compute-intensive vectorizable operations, you can set the appropriate compiler flags and recompile the function code. Then, the compiler automatically vectorizes your code.

For the gcc or clang compiler, add -march=haswell to the command or set -mavx2 as a command option.
To use a specific library, follow instructions in the library's documentation to compile and build the library. For example, to build TensorFlow from source, you can follow the installation instructions on the TensorFlow website. Make sure to use the `--config=mkl` option.

### Enabling AVX2 for Intel MKL

Intel MKL is a library of optimized math operations that implicitly use AVX2 instructions when the compute platform supports them. Frameworks such as PyTorch build with Intel MKL by default, so you don't need to enable AVX2.

Some libraries, such as TensorFlow, provide options in their build process to specify Intel MKL optimization. For example, with TensorFlow, use the `--config=mkl` option.

You can also build popular scientific Python libraries, such as SciPy and NumPy, with Intel MKL. For instructions on building these libraries with Intel MKL, see Numpy/Scipy with Intel MKL and Intel Compilers on the Intel website.

For more information about Intel MKL and similar libraries, see Math Kernel Library in Wikipedia, the OpenBLAS website, and the AMD BLIS repository on GitHub.

### AVX2 support in other languages

If you don't use C or C++ libraries and don't build with Intel MKL, you can still get some AVX2 performance improvement for your applications. Note that the actual improvement depends on the compiler or interpreter's ability to utilize the AVX2 capabilities on your code.

#### Python

Python users generally use SciPy and NumPy libraries for compute-intensive workloads. You can compile these libraries to enable AVX2, or you can use the Intel MKL-enabled versions of the libraries.
Node

For compute-intensive workloads, use AVX2-enabled or Intel MKL-enabled versions of the libraries that you need.

Java

Java's JIT compiler can auto-vectorize your code to run with AVX2 instructions. For information about detecting vectorized code, see the Code vectorization in the JVM presentation on the OpenJDK website.

Go

The standard Go compiler doesn't currently support auto-vectorization, but you can use gccgo, the GCC compiler for Go. Set the -mavx2 option:

```bash
gcc -o avx2 -mavx2 -Wall main.c
```

Intrinsics

It's possible to use intrinsic functions in many languages to manually vectorize your code to use AVX2. However, we don't recommend this approach. Manually writing vectorized code takes significant effort. Also, debugging and maintaining such code is more difficult than using code that depends on auto-vectorization.
Configuring AWS Lambda functions

Learn how to configure the core capabilities and options for your Lambda function using the Lambda API or console. These configurations apply to a function deployed as a container image and for a function deployed as a .zip file archive.

Configuring function options

You can find an overview of how to configure your Lambda function using the console. Includes function memory, timeout, ephemeral storage, advanced logging, triggers, and relational database connections.

Environment variables

You can make your function code portable and keep secrets out of your code by storing them in your function's configuration by using environment variables.

Creating layers

You create a layer to manage your function's dependencies independently and keep your development package small.

Outbound networking

You can use your Lambda function with AWS resources in an Amazon VPC. Connecting your function to a VPC lets you access resources in a private subnet such as relational databases and caches.

Inbound networking

You can use an interface VPC endpoint to invoke your Lambda functions without crossing the public internet.

File system

You can use your Lambda function to mount a Amazon EFS to a local directory. A file system allows your function code to access and modify shared resources safely and at high concurrency.

Aliases

You can configure your clients to invoke a specific Lambda function version by using an alias, instead of updating the client.
**Versions**

By publishing a version of your function, you can store your code and configuration as a separate resource that cannot be changed.

**Response streaming**

You can configure your Lambda function URLs to stream response payloads back to clients. Response streaming can benefit latency sensitive applications by improving time to first byte (TTFB) performance. This is because you can send partial responses back to the client as they become available. Additionally, you can use response streaming to build functions that return larger payloads.
Configuring Lambda function options

After you create a function, you can configure additional capabilities for the function, such as triggers, network access, and file system access. You can also adjust resources associated with the function, such as memory and concurrency. These configurations apply to functions defined as .zip file archives and to functions defined as container images.

You can also create and edit test events to test your function using the console.

For function configuration best practices, see Function configuration.

Sections

- Function versions
- Using the function overview
- Configuring functions (console)
- Configuring function memory (console)
- Accepting function memory recommendations (console)
- Configuring function timeout (console)
- Configuring ephemeral storage (console)
- Configuring triggers (console)
- Configuring Lambda advanced logging options
- Connecting RDS databases (console)

Function versions

A function has an unpublished version, and can have published versions and aliases. By default, the console displays configuration information for the unpublished version of the function. You change the unpublished version when you update your function's code and configuration.

A published version is a snapshot of your function code and configuration that can't be changed (except for a few configuration items relevant to a function version, such as provisioned concurrency).
Using the function overview

The Function overview shows a visualization of your function and its upstream and downstream resources. You can use it to jump to trigger and destination configuration. You can use it to jump to layer configuration for functions defined as .zip file archives.

Configuring functions (console)

For the following function configurations, you can change the settings only for the unpublished version of a function. In the console, the function Configuration tab provides the following sections:

- **General configuration** – Configure memory or opt in to the AWS Compute Optimizer. You can also configure function timeout and the execution role.
- **Permissions** – Configure the execution role and other permissions.
- **Environment variables** – Key-value pairs that Lambda sets in the execution environment. To extend your function's configuration outside of code, use environment variables.
- **Tags** – Key-value pairs that Lambda attaches to your function resource. Use tags to organize Lambda functions into groups for cost reporting and filtering in the Lambda console.

Tags apply to the entire function, including all versions and aliases.

- **Virtual private cloud (VPC)** – If your function needs network access to resources that are not available over the internet, configure it to connect to a virtual private cloud (VPC).
- **Monitoring and operations tools** – configure CloudWatch log groups and other monitoring tools.
• **Concurrency** – Reserve concurrency for a function to set the maximum number of simultaneous executions for a function. Provision concurrency to ensure that a function can scale without fluctuations in latency. Reserved concurrency applies to the entire function, including all versions and aliases.

• **Function URL** – Configure a function URL to add a unique HTTP(S) endpoint to your Lambda function. You can configure a function URL on the $LATEST unpublished function version, or on any function alias.

You can configure the following options on a function, a function version, or an alias.

• **Triggers** – Configure triggers.

• **Destinations** – Configure destinations for asynchronous invocations.

• **Asynchronous invocation** – Configure error handling behavior to reduce the number of retries that Lambda attempts, or the amount of time that unprocessed events stay queued before Lambda discards them. Configure a dead-letter queue to retain discarded events.

• **Code signing** – To use Code signing with your function, configure the function to include a code-signing configuration.

• **File systems** – Connect your function to a file system.

• **State machines** – Use a state machine to orchestrate and apply error handling to your function.

The console provides separate tabs to configure aliases and versions:

• **Aliases** – An alias is a named resource that maps to a function version. You can change an alias to map to a different function version.

• **Versions** – Lambda assigns a new version number each time you publish your function. For more information about managing versions, see Lambda function versions.

You can configure the following items for a published function version:

• Triggers

• Destinations

• Provisioned concurrency

• Asynchronous invocation

• Database connections and proxies
Configuring function memory (console)

Lambda allocates CPU power in proportion to the amount of memory configured. Memory is the amount of memory available to your Lambda function at runtime. You can increase or decrease the memory and CPU power allocated to your function using the Memory (MB) setting. To configure the memory for your function, set a value between 128 MB and 10,240 MB in 1-MB increments. At 1,769 MB, a function has the equivalent of one vCPU (one vCPU-second of credits per second).

You can configure the memory of your function in the Lambda console.

Accepting function memory recommendations (console)

If you have administrator permissions in AWS Identity and Access Management (IAM), you can opt in to receive Lambda function memory setting recommendations from AWS Compute Optimizer. For instructions on opting in to memory recommendations for your account or organization, see Opting in your account in the AWS Compute Optimizer User Guide.

Note
Compute Optimizer supports only functions that use x86_64 architecture.

When you've opted in and your Lambda function meets Compute Optimizer requirements, you can view and accept function memory recommendations from Compute Optimizer in the Lambda console in General configuration.

Configuring function timeout (console)

Lambda runs your code for a set amount of time before timing out. Timeout is the maximum amount of time in seconds that a Lambda function can run. The default value for this setting is 3 seconds, but you can adjust this in increments of 1 second up to a maximum value of 15 minutes.

Configuring ephemeral storage (console)

By default, Lambda allocates 512 MB for a function’s /tmp directory. You can increase or decrease this amount using the Ephemeral storage (MB) setting. To configure the size of a function’s /tmp directory, set a whole number value between 512 MB and 10,240 MB, in 1-MB increments.
Note

Configuring ephemeral storage past the default 512 MB allocated incurs a cost. For more information, see Lambda pricing.

Configuring triggers (console)

You can configure other AWS services to trigger your function each time a specified event occurs. For details about how services trigger Lambda functions, see Using AWS Lambda with other services.

Configuring Lambda advanced logging options

To give you more control over how your functions’ logs are captured, processed, and consumed, Lambda offers the following logging configuration options:

- **Log format** - select between plain text and structured JSON format for your function’s logs
- **Log level** - for JSON structured logs, choose the detail level of the logs Lambda sends to CloudWatch, such as ERROR, DEBUG, or INFO
- **Log group** - choose the CloudWatch log group your function sends logs to

For more information about configuring these options in the Lambda console, see the section called “Configuring advanced logging controls for your Lambda function”.

Connecting RDS databases (console)

You can connect a Lambda function to an Amazon Relational Database Service (Amazon RDS) database directly and through an Amazon RDS Proxy. Direct connections are useful in simple scenarios, and proxies are recommended for production. A database proxy manages a pool of shared database connections which enables your function to reach high concurrency levels without exhausting database connections.

To connect Lambda functions and Amazon RDS, see Using AWS Lambda with Amazon RDS
Using Lambda environment variables

You can use environment variables to adjust your function's behavior without updating code. An environment variable is a pair of strings that is stored in a function's version-specific configuration. The Lambda runtime makes environment variables available to your code and sets additional environment variables that contain information about the function and invocation request.

### Note

To increase database security, we recommend that you use AWS Secrets Manager instead of environment variables to store database credentials. For more information, see [Using AWS Lambda with Amazon RDS](#).

Environment variables are not evaluated before the function invocation. Any value you define is considered a literal string and not expanded. Perform the variable evaluation in your function code.

### Sections

- Configuring environment variables
- Configuring environment variables with the API
- Example scenario for environment variables
- Retrieve environment variables
- Defined runtime environment variables
- Securing environment variables
- Sample code and templates

### Configuring environment variables

You define environment variables on the unpublished version of your function. When you publish a version, the environment variables are locked for that version along with other version-specific configuration.

You create an environment variable for your function by defining a key and a value. Your function uses the name of the key to retrieve the value of the environment variable.
To set environment variables in the Lambda console

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose Configuration, then choose Environment variables.
5. Choose Add environment variable.
6. Enter a key and value.

Requirements

- Keys start with a letter and are at least two characters.
- Keys only contain letters, numbers, and the underscore character (_).
- Keys aren't reserved by Lambda.
- The total size of all environment variables doesn't exceed 4 KB.
7. Choose Save.

To generate a list of environment variables in the console code editor

You can generate a list of environment variables in the Lambda code editor. This is a quick way to reference your environment variables while you code.

1. Choose the Code tab.
2. Choose the Environment Variables tab.
3. Choose Tools, Show Environment Variables.

Environment variables remain encrypted when listed in the console code editor. If you enabled encryption helpers for encryption in transit, then those settings remain unchanged. For more information, see Securing environment variables.

The environment variables list is read-only and is available only on the Lambda console. This file is not included when you download the function's .zip file archive, and you can't add environment variables by uploading this file.
Configuring environment variables with the API

To manage environment variables with the AWS CLI or AWS SDK, use the following API operations.

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

The following example sets two environment variables on a function named my-function.

```
aws lambda update-function-configuration --function-name my-function
   --environment "Variables={BUCKET=my-bucket,KEY=file.txt}"
```

When you apply environment variables with the update-function-configuration command, the entire contents of the Variables structure is replaced. To retain existing environment variables when you add a new one, include all existing values in your request.

To get the current configuration, use the get-function-configuration command.

```
aws lambda get-function-configuration --function-name my-function
```

You should see the following output:

```
{
   "FunctionName": "my-function",
   "Runtime": "nodejs20.x",
   "Role": "arn:aws:iam::123456789012:role/lambda-role",
   "Environment": {
      "Variables": {
         "BUCKET": "my-bucket",
         "KEY": "file.txt"
      }
   },
   "RevisionId": "0894d3c1-2a3d-4d48-bf7f-aba9f3c15",
   ...
}
```
You can pass the revision ID from the output of `get-function-configuration` as a parameter to `update-function-configuration`. This ensures that the values don't change between when you read the configuration and when you update it.

To configure a function's encryption key, set the `KMSKeyARN` option.

```
aws lambda update-function-configuration --function-name my-function \
   --kms-key-arn arn:aws:kms:us-east-2:123456789012:key/055efbb4-xmpl-4336-ba9c-538c7d31f599
```

**Example scenario for environment variables**

You can use environment variables to customize function behavior in your test environment and production environment. For example, you can create two functions with the same code but different configurations. One function connects to a test database, and the other connects to a production database. In this situation, you use environment variables to pass the hostname and other connection details for the database to the function.

The following example shows how to define the database host and database name as environment variables.

```
<table>
<thead>
<tr>
<th>ENVIRONMENT</th>
<th>DEVELOPMENT</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>databaseHost</td>
<td>lambdaadb</td>
<td>Remove</td>
</tr>
<tr>
<td>databaseName</td>
<td>rd1owwlydymnm5.cuovuyfg08\</td>
<td>Remove</td>
</tr>
<tr>
<td>Key</td>
<td>Value</td>
<td></td>
</tr>
</tbody>
</table>
```

If you want your test environment to generate more debug information than the production environment, you could set an environment variable to configure your test environment to use more verbose logging or more detailed tracing.

**Retrieve environment variables**

To retrieve environment variables in your function code, use the standard method for your programming language.
Node.js

```javascript
let region = process.env.AWS_REGION
```

Python

```python
import os
region = os.environ['AWS_REGION']
```

**Note**

In some cases, you may need to use the following format:

```python
region = os.environ.get('AWS_REGION')
```

Ruby

```ruby
region = ENV['AWS_REGION']
```

Java

```java
String region = System.getenv("AWS_REGION");
```

Go

```go
var region = os.Getenv("AWS_REGION")
```

C#

```csharp
string region = Environment.GetEnvironmentVariable("AWS_REGION");
```

PowerShell

```powershell
$region = $env:AWS_REGION
```
Lambda stores environment variables securely by encrypting them at rest. You can configure Lambda to use a different encryption key, encrypt environment variable values on the client side, or set environment variables in an AWS CloudFormation template with AWS Secrets Manager.

**Defined runtime environment variables**

Lambda runtimes set several environment variables during initialization. Most of the environment variables provide information about the function or runtime. The keys for these environment variables are reserved and cannot be set in your function configuration.

**Reserved environment variables**

- `_HANDLER` – The handler location configured on the function.
- `_X_AMZN_TRACE_ID` – The X-Ray tracing header. This environment variable changes with each invocation.
  - This environment variable is not defined for OS-only runtimes (the provided runtime family). You can set `_X_AMZN_TRACE_ID` for custom runtimes using the Lambda-Runtime-Trace-Id response header from the Next invocation.
  - For Java runtime versions 17 and later, this environment variable is not used. Instead, Lambda stores tracing information in the `com.amazonaws.xray.traceHeader` system property.
- `AWS_DEFAULT_REGION` – The default AWS Region where the Lambda function is executed.
- `AWS_REGION` – The AWS Region where the Lambda function is executed. If defined, this value overrides the `AWS_DEFAULT_REGION`.
- `AWS_EXECUTION_ENV` – The runtime identifier, prefixed by `AWS_Lambda_` (for example, `AWS_Lambda_java8`). This environment variable is not defined for OS-only runtimes (the provided runtime family).
- `AWS_LAMBDA_FUNCTION_NAME` – The name of the function.
- `AWS_LAMBDA_FUNCTION_MEMORY_SIZE` – The amount of memory available to the function in MB.
- `AWS_LAMBDA_FUNCTION_VERSION` – The version of the function being executed.
- `AWS_LAMBDA_INITIALIZATION_TYPE` – The initialization type of the function, which is on-demand, provisioned-concurrency, or snap-start. For information, see Configuring provisioned concurrency or Improving startup performance with Lambda SnapStart.
- `AWS_LAMBDA_LOG_GROUP_NAME`, `AWS_LAMBDA_LOG_STREAM_NAME` – The name of the Amazon CloudWatch Logs group and stream for the function. The `AWS_LAMBDA_LOG_GROUP_NAME` and
AWS.lambda_log_stream_name environment variables are not available in Lambda SnapStart functions.

- AWS_ACCESS_KEY, AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, AWS_SESSION_TOKEN – The access keys obtained from the function's execution role.
- AWS.LAMBDA_RUNTIME_API – (Custom runtime) The host and port of the runtime API.
- LAMBDA_TASK_ROOT – The path to your Lambda function code.
- LAMBDA_RUNTIME_DIR – The path to runtime libraries.

The following additional environment variables aren't reserved and can be extended in your function configuration.

Unreserved environment variables

- LANG – The locale of the runtime (en_US.UTF-8).
- PATH – The execution path (/usr/local/bin:/usr/bin:/bin:/opt/bin).
- LD_LIBRARY_PATH – The system library path (/lib64:/usr/lib64:
  $LAMBDA_RUNTIME_DIR:$LAMBDA_RUNTIME_DIR/lib:$LAMBDA_TASK_ROOT:
  $LAMBDA_TASK_ROOT/lib:/opt/lib):
- NODE_PATH – (Node.js) The Node.js library path (/opt/nodejs/node12/node_modules:/
  opt/nodejs/node_modules:$LAMBDA_RUNTIME_DIR/node_modules).
- PYTHONPATH – (Python 2.7, 3.6, 3.8) The Python library path ($LAMBDA_RUNTIME_DIR).
- GEM_PATH – (Ruby) The Ruby library path ($LAMBDA_TASK_ROOT/vendor/bundle/
  ruby/2.5.0:/opt/ruby/gems/2.5.0).
- AWS_XRAY_CONTEXT_MISSING – For X-Ray tracing, Lambda sets this to LOG_ERROR to avoid
  throwing runtime errors from the X-Ray SDK.
- AWS_XRAY_DAEMON_ADDRESS – For X-Ray tracing, the IP address and port of the X-Ray daemon.
- AWS.LAMBDA_DOTNET_PREJIT – For the .NET 6 and .NET 7 runtimes, set this variable to enable or disable .NET specific runtime optimizations. Values include always, never, and provisioned-concurrency. For more information, see Configuring provisioned concurrency.
- TZ – The environment's time zone (UTC). The execution environment uses NTP to synchronize the system clock.

The sample values shown reflect the latest runtimes. The presence of specific variables or their values can vary on earlier runtimes.
Securing environment variables

For securing your environment variables, you can use server-side encryption to protect your data at rest and client-side encryption to protect your data in transit.

ℹ️ Note

To increase database security, we recommend that you use AWS Secrets Manager instead of environment variables to store database credentials. For more information, see Using AWS Lambda with Amazon RDS.

Security at rest

Lambda always provides server-side encryption at rest with an AWS KMS key. By default, Lambda uses an AWS managed key. If this default behavior suits your workflow, you don't need to set up anything else. Lambda creates the AWS managed key in your account and manages permissions to it for you. AWS doesn't charge you to use this key.

If you prefer, you can provide an AWS KMS customer managed key instead. You might do this to have control over rotation of the KMS key or to meet the requirements of your organization for managing KMS keys. When you use a customer managed key, only users in your account with access to the KMS key can view or manage environment variables on the function.

Customer managed keys incur standard AWS KMS charges. For more information, see AWS Key Management Service pricing.

Security in transit

For additional security, you can enable helpers for encryption in transit, which ensures that your environment variables are encrypted client-side for protection in transit.

To configure encryption for your environment variables

1. Use the AWS Key Management Service (AWS KMS) to create any customer managed keys for Lambda to use for server-side and client-side encryption. For more information, see Creating keys in the AWS Key Management Service Developer Guide.
2. Using the Lambda console, navigate to the Edit environment variables page.
   a. Open the Functions page of the Lambda console.
b. Choose a function.

c. Choose Configuration, then choose Environment variables from the left navigation bar.

d. In the Environment variables section, choose Edit.

e. Expand Encryption configuration.

3. (Optional) Enable console encryption helpers to use client-side encryption to protect your data in transit.


   b. For each environment variable that you want to enable console encryption helpers for, choose Encrypt next to the environment variable.

   c. Under AWS KMS key to encrypt in transit, choose a customer managed key that you created at the beginning of this procedure.

   d. Choose Execution role policy and copy the policy. This policy grants permission to your function's execution role to decrypt the environment variables.

      Save this policy to use in the last step of this procedure.

   e. Add code to your function that decrypts the environment variables. To see an example, choose Decrypt secrets snippet.

4. (Optional) Specify your customer managed key for encryption at rest.

   a. Choose Use a customer master key.

   b. Choose a customer managed key that you created at the beginning of this procedure.

5. Choose Save.

6. Set up permissions.

   If you're using a customer managed key with server-side encryption, grant permissions to any users or roles that you want to be able to view or manage environment variables on the function. For more information, see Managing permissions to your server-side encryption KMS key.

   If you're enabling client-side encryption for security in transit, your function needs permission to call the kms:Decrypt API operation. Add the policy that you saved previously in this procedure to the function's execution role.
Managing permissions to your server-side encryption KMS key

No AWS KMS permissions are required for your user or the function's execution role to use the default encryption key. To use a customer managed key, you need permission to use the key. Lambda uses your permissions to create a grant on the key. This allows Lambda to use it for encryption.

- `kms:ListAliases` – To view keys in the Lambda console.
- `kms:CreateGrant`, `kms:Encrypt` – To configure a customer managed key on a function.
- `kms:Decrypt` – To view and manage environment variables that are encrypted with a customer managed key.

You can get these permissions from your AWS account or from a key's resource-based permissions policy. `ListAliases` is provided by the managed policies for Lambda. Key policies grant the remaining permissions to users in the Key users group.

Users without ` Decrypt` permissions can still manage functions, but they can't view environment variables or manage them in the Lambda console. To prevent a user from viewing environment variables, add a statement to the user's permissions that denies access to the default key, a customer managed key, or all keys.

**Example IAM policy – Deny access by key ARN**

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Deny",
            "Action": ["kms:Decrypt"],
            "Resource": "arn:aws:kms:us-east-2:123456789012:key/3be10e2d-xmpl-4be4-bc9d-0405a71945cc"
        }
    ]
}
```
For details on managing key permissions, see Key policies in AWS KMS in the AWS Key Management Service Developer Guide.

Sample code and templates

Sample applications in this guide's GitHub repository demonstrate the use of environment variables in function code and AWS CloudFormation templates.

Sample applications

- **Blank function** – Create a basic function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests, and the AWS SDK.
- **RDS MySQL** – Create a VPC and an Amazon Relational Database Service (Amazon RDS) DB instance in one template, with a password stored in Secrets Manager. In the application template, import database details from the VPC stack, read the password from Secrets Manager, and pass all connection configuration to the function in environment variables.
Connecting outbound networking to resources in a VPC

You can configure a Lambda function to connect to private subnets in a virtual private cloud (VPC) in your AWS account. Use Amazon Virtual Private Cloud (Amazon VPC) to create a private network for resources such as databases, cache instances, or internal services. Connect your function to the VPC to access private resources while the function is running. This section provides a summary of Lambda VPC connections. For details about VPC networking in Lambda, see the section called “Private networking”.

Tip
To configure your Lambda function to access a VPC and subnet, you can use the Lambda Console or the API.
Refer to the VpcConfig section in CreateFunction to configure your function. See Configuring VPC access (console) and Configuring VPC access (API) for detailed steps.

When you connect a function to a VPC, Lambda assigns your function to a Hyperplane ENI (elastic network interface) for each subnet in your function's VPC configuration. Lambda creates a Hyperplane ENI the first time a unique subnet and security group combination is defined for a VPC-enabled function in an account.

While Lambda creates a Hyperplane ENI, you can't perform additional operations that target the function, such as creating versions or updating the function's code. For new functions, you can't invoke the function until its state changes from Pending to Active. For existing functions, you can still invoke an earlier version while the update is in progress. For details about the Hyperplane ENI lifecycle, see the section called “Lambda Hyperplane ENIs”.

Lambda functions can't connect directly to a VPC with dedicated instance tenancy. To connect to resources in a dedicated VPC, peer it to a second VPC with default tenancy.

Sections
- Managing VPC connections
- Execution role and user permissions
- Configuring VPC access (console)
- Configuring VPC access (API)
- Using IAM condition keys for VPC settings
Managing VPC connections

Multiple functions can share a network interface, if the functions share the same subnet and security group. Connecting additional functions to the same VPC configuration (subnet and security group) that has an existing Lambda-managed network interface is much quicker than creating a new network interface.

If your functions aren't active for a long period of time, Lambda reclaims its network interfaces, and the functions become Idle. To reactivate an idle function, invoke it. This invocation fails, and the function enters a Pending state again until a network interface is available.

If you update your function to access a different VPC, it terminates connectivity from the Hyperplane ENI to the previous VPC. The process to update the connectivity to a new VPC can take several minutes. During this time, Lambda connects function invocations to the previous VPC. After the update is complete, new invocations start using the new VPC and the Lambda function is no longer connected to the older VPC.

For short-lived operations, such as DynamoDB queries, the latency overhead of setting up a TCP connection might be greater than the operation itself. To ensure connection reuse for short-lived/infrequently invoked functions, we recommend that you use TCP keep-alive for connections that were created during your function initialization, to avoid creating new connections for subsequent invokes. For more information on reusing connections using keep-alive, refer to Lambda documentation on reusing connections.

Execution role and user permissions

Lambda uses your function's permissions to create and manage network interfaces. To connect to a VPC, your function's execution role must have the following permissions:

Execution role permissions

- `ec2:CreateNetworkInterface`
- `ec2:DescribeNetworkInterfaces` – This action only works if it's allowed on all resources ("Resource": "*").
• **ec2:DeleteNetworkInterface** – If you don't specify a resource ID for **DeleteNetworkInterface** in the execution role, your function may not be able to access the VPC. Either specify a unique resource ID, or include all resource IDs, for example, "Resource": "arn:aws:ec2:us-west-2:123456789012:*/*".

• **ec2:AssignPrivateIpAddresses**

• **ec2:UnassignPrivateIpAddresses**

These permissions are included in the AWS managed policy **AWSLambdaVPCAccessExecutionRole**. Note that these permissions are required only to create ENIs, not to invoke your VPC function. In other words, you are still able to invoke your VPC function successfully even if you remove these permissions from your execution role. To completely disassociate your Lambda function from the VPC, update the function's VPC configuration settings using the console or the **UpdateFunctionConfiguration** API.

When you configure VPC connectivity, Lambda uses your permissions to verify network resources. To configure a function to connect to a VPC, your user needs the following permissions:

**User permissions**

• **ec2:DescribeSecurityGroups**

• **ec2:DescribeSubnets**

• **ec2:DescribeVpcs**

**Configuring VPC access (console)**

If your **IAM permissions** allow you only to create Lambda functions that connect to your VPC, you must configure the VPC when you create the function. If your IAM permissions allow you to create functions that aren't connected to your VPC, you can add the VPC configuration after you create the function.

**To configure a VPC when you create a function**

1. Open the **Functions page** of the Lambda console.
2. Choose **Create function**.
3. Under **Basic information**, for **Function name**, enter a name for your function.
4. Expand **Advanced settings**.
5. Select **Enable VPC**, and then choose the VPC that you want the function to access.

6. (Optional) To allow **outbound IPv6 traffic**, select **Allow IPv6 traffic for dual-stack subnets**.

7. Choose subnets and security groups. If you selected **Allow IPv6 traffic for dual-stack subnets**, all selected subnets must have an IPv4 CIDR block and an IPv6 CIDR block.

8. Choose **Create function**.

---

**To configure a VPC for an existing function**

1. Open the [Functions page](https://console.aws.amazon.com/lambda/) of the Lambda console.

2. Choose a function.

3. Choose **Configuration** and then choose **VPC**.

4. Under **VPC**, choose **Edit**.

5. Choose the VPC that you want the function to access.

6. (Optional) To allow **outbound IPv6 traffic**, select **Allow IPv6 traffic for dual-stack subnets**.

7. Choose subnets and security groups. If you selected **Allow IPv6 traffic for dual-stack subnets**, all selected subnets must have an IPv4 CIDR block and an IPv6 CIDR block.

8. Choose **Save**.

---

**Configuring VPC access (API)**

To connect a Lambda function to a VPC, you can use the following API operations:
To create a function and connect it to a VPC using the AWS Command Line Interface (AWS CLI), you can use the `create-function` command with the `vpc-config` option. The following example creates a function with a VPC connection. The function has access to two subnets and one security group and allows **outbound IPv6 traffic**.

```bash
aws lambda create-function --function-name my-function
  --runtime nodejs20.x --handler index.js --zip-file fileb://function.zip
  --role arn:aws:iam::123456789012:role/lambda-role
  --vpc-config Ipv6AllowedForDualStack=true,SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a036,SecurityGroupIds=sg-085912345678492fb
```

To connect an existing function to a VPC, use the `update-function-configuration` command with the `vpc-config` option.

```bash
aws lambda update-function-configuration --function-name my-function
  --vpc-config SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a036,SecurityGroupIds=sg-085912345678492fb
```

To disconnect your function from a VPC, update the function configuration with an empty list of subnets and security groups.

```bash
aws lambda update-function-configuration --function-name my-function
  --vpc-config SubnetIds=[],SecurityGroupIds=[]
```

### Using IAM condition keys for VPC settings

You can use Lambda-specific condition keys for VPC settings to provide additional permission controls for your Lambda functions. For example, you can require that all functions in your organization are connected to a VPC. You can also specify the subnets and security groups that the function's users can and can't use.

Lambda supports the following condition keys in IAM policies:

- **lambda:VpcIds** – Allow or deny one or more VPCs.
- **lambda:SubnetIds** – Allow or deny one or more subnets.
• **lambda:SecurityGroupIds** – Allow or deny one or more security groups.


**Tip**

If your function already includes a VPC configuration from a previous API request, you can send an `UpdateFunctionConfiguration` request without the VPC configuration.

**Example policies with condition keys for VPC settings**

The following examples demonstrate how to use condition keys for VPC settings. After you create a policy statement with the desired restrictions, append the policy statement for the target user or role.

**Ensure that users deploy only VPC-connected functions**

To ensure that all users deploy only VPC-connected functions, you can deny function create and update operations that don't include a valid VPC ID.

Note that VPC ID is not an input parameter to the `CreateFunction` or `UpdateFunctionConfiguration` request. Lambda retrieves the VPC ID value based on the subnet and security group parameters.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceVPCFunction",
      "Action": [
        "lambda:CreateFunction",
        "lambda:UpdateFunctionConfiguration"
      ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "Null": {
        }
      }
    }
  ]
}
```
Deny users access to specific VPCs, subnets, or security groups

To deny users access to specific VPCs, use StringEquals to check the value of the `lambda:VpcIds` condition. The following example denies users access to vpc-1 and vpc-2.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EnforceOutOfVPC",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Effect": "Deny",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "lambda:VpcIds": ["vpc-1", "vpc-2"]
                }
            }
        }
    ]
}
```

To deny users access to specific subnets, use StringEquals to check the value of the `lambda:SubnetIds` condition. The following example denies users access to subnet-1 and subnet-2.

```
{
    "Sid": "EnforceOutOfSubnet",
    "Action": [
        "lambda:CreateFunction",
        "lambda:UpdateFunctionConfiguration"
    ],
    "Effect": "Deny",
    "Resource": "*",
    "Condition": {
        "StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
    }
}
```
To deny users access to specific security groups, use `StringEquals` to check the value of the `lambda:SecurityGroupIds` condition. The following example denies users access to `sg-1` and `sg-2`.

```json
{
    "Sid": "EnforceOutOfSecurityGroups",
    "Action": [
        "lambda:CreateFunction",
        "lambda:UpdateFunctionConfiguration"
    ],
    "Effect": "Deny",
    "Resource": "*",
    "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
    }
}
```

**Allow users to create and update functions with specific VPC settings**

To allow users to access specific VPCs, use `StringEquals` to check the value of the `lambda:VpcIds` condition. The following example allows users to access `vpc-1` and `vpc-2`.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EnforceStayInSpecificVpc",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Effect": "Allow",
            "Resource": "*",
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "lambda:VpcIds": ["vpc-1", "vpc-2"]
                }
            }
        }
    ]
}
```
To allow users to access specific subnets, use StringEquals to check the value of the lambda:SubnetIds condition. The following example allows users to access subnet-1 and subnet-2.

```
{
    "Sid": "EnforceStayInSpecificSubnets",
    "Action": [
        "lambda:CreateFunction",
        "lambda:UpdateFunctionConfiguration"
    ],
    "Effect": "Allow",
    "Resource": "*",
    "Condition": {
        "StringEquals": {
            "lambda:VpcIds": ["vpc-1", "vpc-2"]
        }
    }
}
```

To allow users to access specific security groups, use StringEquals to check the value of the lambda:SecurityGroupIds condition. The following example allows users to access sg-1 and sg-2.

```
{
    "Sid": "EnforceStayInSpecificSecurityGroup",
    "Action": [
        "lambda:CreateFunction",
        "lambda:UpdateFunctionConfiguration"
    ],
    "Effect": "Allow",
    "Resource": "*",
    "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
    }
}
```
Internet and service access for VPC-connected functions

By default, Lambda runs your functions in a secure VPC with access to AWS services and the internet. Lambda owns this VPC, which isn't connected to your account's default VPC. When you connect a function to a VPC in your account, the function can't access the internet unless your VPC provides access.

Note

Several AWS services offer VPC endpoints. You can use VPC endpoints to connect to AWS services from within a VPC without internet access.

Internet access from a private subnet requires network address translation (NAT). To give your function access to the internet, route outbound traffic to a NAT gateway in a public subnet. The NAT gateway has a public IP address and can connect to the internet through the VPC's internet gateway. An idle NAT gateway connection will time out after 350 seconds. For more information, see How do I give internet access to my Lambda function in a VPC?

VPC tutorials

In the following tutorials, you connect a Lambda function to resources in your VPC.

- Tutorial: Using a Lambda function to access Amazon RDS in an Amazon VPC
- Tutorial: Configuring a Lambda function to access Amazon ElastiCache in an Amazon VPC
Sample VPC configurations

You can use the following sample AWS CloudFormation templates to create VPC configurations to use with Lambda functions. There are two templates available in this guide's GitHub repository:

- **vpc-private.yaml** – A VPC with two private subnets and VPC endpoints for Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB. Use this template to create a VPC for functions that don't need internet access. This configuration supports use of Amazon S3 and DynamoDB with the AWS SDKs, and access to database resources in the same VPC over a local network connection.

- **vpc-privatepublic.yaml** – A VPC with two private subnets, VPC endpoints, a public subnet with a NAT gateway, and an internet gateway. Internet-bound traffic from functions in the private subnets is routed to the NAT gateway using a route table.

To create a VPC using a template, on the AWS CloudFormation console [Stacks page](#), choose Create stack, and then follow the instructions in the Create stack wizard.
Connecting inbound interface VPC endpoints for Lambda

If you use Amazon Virtual Private Cloud (Amazon VPC) to host your AWS resources, you can establish a connection between your VPC and Lambda. You can use this connection to invoke your Lambda function without crossing the public internet.

To establish a private connection between your VPC and Lambda, create an **interface VPC endpoint**. Interface endpoints are powered by **AWS PrivateLink**, which enables you to privately access Lambda APIs without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don’t need public IP addresses to communicate with Lambda APIs. Traffic between your VPC and Lambda does not leave the AWS network.

Each interface endpoint is represented by one or more **elastic network interfaces** in your subnets. A network interface provides a private IP address that serves as an entry point for traffic to Lambda.

**Sections**
- Considerations for Lambda interface endpoints
- Creating an interface endpoint for Lambda
- Creating an interface endpoint policy for Lambda

**Considerations for Lambda interface endpoints**

Before you set up an interface endpoint for Lambda, be sure to review **Interface endpoint properties and limitations** in the *Amazon VPC User Guide*.

You can call any of the Lambda API operations from your VPC. For example, you can invoke the Lambda function by calling the **Invoke** API from within your VPC. For the full list of Lambda APIs, see **Actions** in the Lambda API reference.

**use1-az3** is a limited capacity Region for Lambda VPC functions. You shouldn’t use subnets in this availability zone with your Lambda functions because this can result in reduced zonal redundancy in the event of an outage.

**Keep-alive for persistent connections**

Lambda purges idle connections over time, so you must use a keep-alive directive to maintain persistent connections. Attempting to reuse an idle connection when invoking a function results in a connection error. To maintain your persistent connection, use the keep-alive directive associated
with your runtime. For an example, see Reusing Connections with Keep-Alive in Node.js in the AWS SDK for JavaScript Developer Guide.

Billing Considerations

There is no additional cost to access a Lambda function through an interface endpoint. For more Lambda pricing information, see AWS Lambda Pricing.

Standard pricing for AWS PrivateLink applies to interface endpoints for Lambda. Your AWS account is billed for every hour an interface endpoint is provisioned in each Availability Zone and for data processed through the interface endpoint. For more interface endpoint pricing information, see AWS PrivateLink pricing.

VPC Peering Considerations

You can connect other VPCs to the VPC with interface endpoints using VPC peering. VPC peering is a networking connection between two VPCs. You can establish a VPC peering connection between your own two VPCs, or with a VPC in another AWS account. The VPCs can also be in two different AWS Regions.

Traffic between peered VPCs stays on the AWS network and does not traverse the public internet. Once VPCs are peered, resources like Amazon Elastic Compute Cloud (Amazon EC2) instances, Amazon Relational Database Service (Amazon RDS) instances, or VPC-enabled Lambda functions in both VPCs can access the Lambda API through interface endpoints created in the one of the VPCs.

Creating an interface endpoint for Lambda

You can create an interface endpoint for Lambda using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see Creating an interface endpoint in the Amazon VPC User Guide.

To create an interface endpoint for Lambda (console)

1. Open the Endpoints page of the Amazon VPC console.
2. Choose Create Endpoint.
3. For Service category, verify that AWS services is selected.
4. For Service Name, choose com.amazonaws.region.lambda. Verify that the Type is Interface.
5. Choose a VPC and subnets.
6. To enable private DNS for the interface endpoint, select the **Enable DNS Name** check box.

7. For **Security group**, choose one or more security groups.

8. Choose **Create endpoint**.

To use the private DNS option, you must set the `enableDnsHostnames` and `enableDnsSupport` attributes of your VPC. For more information, see Viewing and updating DNS support for your VPC in the Amazon VPC User Guide. If you enable private DNS for the interface endpoint, you can make API requests to Lambda using its default DNS name for the Region, for example, `lambda.us-east-1.amazonaws.com`. For more service endpoints, see Service endpoints and quotas in the AWS General Reference.

For more information, see Accessing a service through an interface endpoint in the Amazon VPC User Guide.

For information about creating and configuring an endpoint using AWS CloudFormation, see the AWS::EC2::VPCEndpoint resource in the AWS CloudFormation User Guide.

**To create an interface endpoint for Lambda (AWS CLI)**

Use the create-vpc-endpoint command and specify the VPC ID, VPC endpoint type (interface), service name, subnets that will use the endpoint, and security groups to associate with the endpoint's network interfaces. For example:

```bash
aws ec2 create-vpc-endpoint --vpc-id vpc-ec43eb89 --vpc-endpoint-type Interface --service-name com.amazonaws.us-east-1.lambda --subnet-id subnet-abababab --security-group-id sg-1a2b3c4d
```

**Creating an interface endpoint policy for Lambda**

To control who can use your interface endpoint and which Lambda functions the user can access, you can attach an endpoint policy to your endpoint. The policy specifies the following information:

- The principal that can perform actions.
- The actions that the principal can perform.
- The resources on which the principal can perform actions.
For more information, see [Controlling access to services with VPC endpoints](https://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/vpc Endpoints.html) in the *Amazon VPC User Guide*.

**Example: Interface endpoint policy for Lambda actions**

The following is an example of an endpoint policy for Lambda. When attached to an endpoint, this policy allows user MyUser to invoke the function *my-function*.

> **Note**
> You need to include both the qualified and the unqualified function ARN in the resource.

```json
{
    "Statement": [
        {
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:user/MyUser"
            },
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": [
            ]
        }
    ]
}
```
Configuring file system access for Lambda functions

You can configure a function to mount an Amazon Elastic File System (Amazon EFS) file system to a local directory. With Amazon EFS, your function code can access and modify shared resources safely and at high concurrency.

Sections

• Execution role and user permissions
• Configuring a file system and access point
• Connecting to a file system (console)
• Configuring file system access with the Lambda API
• Mounting an Amazon EFS file system in another AWS account
• AWS CloudFormation and AWS SAM
• Sample applications

Execution role and user permissions

If the file system doesn't have a user-configured AWS Identity and Access Management (IAM) policy, EFS uses a default policy that grants full access to any client that can connect to the file system using a file system mount target. If the file system has a user-configured IAM policy, your function's execution role must have the correct elasticfilesystem permissions.

Execution role permissions

• elasticfilesystem:ClientMount
• elasticfilesystem:ClientWrite (not required for read-only connections)

These permissions are included in the AmazonElasticFileSystemClientReadWriteAccess managed policy. Additionally, your execution role must have the permissions required to connect to the file system's VPC.

When you configure a file system, Lambda uses your permissions to verify mount targets. To configure a function to connect to a file system, your user needs the following permissions:

User permissions

• elasticfilesystem:DescribeMountTargets
Configuring a file system and access point

Create a file system in Amazon EFS with a mount target in every Availability Zone that your function connects to. For performance and resilience, use at least two Availability Zones. For example, in a simple configuration you could have a VPC with two private subnets in separate Availability Zones. The function connects to both subnets and a mount target is available in each. Ensure that NFS traffic (port 2049) is allowed by the security groups used by the function and mount targets.

Note

When you create a file system, you choose a performance mode that can't be changed later. **General purpose** mode has lower latency, and **Max I/O** mode supports a higher maximum throughput and IOPS. For help choosing, see [Amazon EFS performance](https://docs.aws.amazon.com/efs/latest/ug/) in the *Amazon Elastic File System User Guide*.

An access point connects each instance of the function to the right mount target for the Availability Zone it connects to. For best performance, create an access point with a non-root path, and limit the number of files that you create in each directory. The following example creates a directory named `my-function` on the file system and sets the owner ID to 1001 with standard directory permissions (755).

**Example access point configuration**

- **Name** – files
- **User ID** – 1001
- **Group ID** – 1001
- **Path** – `/my-function`
- **Permissions** – 755
- **Owner user ID** – 1001
- **Group user ID** – 1001

When a function uses the access point, it is given user ID 1001 and has full access to the directory.

For more information, see the following topics in the *Amazon Elastic File System User Guide*:
Connecting to a file system (console)

A function connects to a file system over the local network in a VPC. The subnets that your function connects to can be the same subnets that contain mount points for your file system, or subnets in the same Availability Zone that can route NFS traffic (port 2049) to the file system.

Note

If your function is not already connected to a VPC, see Connecting outbound networking to resources in a VPC.

To configure file system access

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose Configuration and then choose File systems.
4. Under File system, choose Add file system.
5. Configure the following properties:
   - **EFS file system** – The access point for a file system in the same VPC.
   - **Local mount path** – The location where the file system is mounted on the Lambda function, starting with /mnt/.

Pricing

Amazon EFS charges for storage and throughput, with rates that vary by storage class. For details, see Amazon EFS pricing.

Lambda charges for data transfer between VPCs. This only applies if your function's VPC is peered to another VPC with a file system. The rates are the same as for Amazon EC2 data transfer between VPCs in the same Region. For details, see Lambda pricing.
For more information about Lambda's integration with Amazon EFS, see Using Amazon EFS with Lambda.

**Configuring file system access with the Lambda API**

Use the following API operations to connect your Lambda function to a file system:

- CreateFunction
- UpdateFunctionConfiguration

To connect a function to a file system, use the `update-function-configuration` command. The following example connects a function named `my-function` to a file system with ARN of an access point.

```
ARN=arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd
aws lambda update-function-configuration --function-name my-function --file-system-configs Arn=$ARN,LocalMountPath=/mnt/efs0
```

You can get the ARN of a file system's access point with the `describe-access-points` command.

```
aws efs describe-access-points
```

You should see the following output:

```
{
    "AccessPoints": [
        {
            "ClientToken": "console-aa50c1fd-xmpl-48b5-91ce-57b27a3b1017",
            "Name": "lambda-ap",
            "Tags": [
                {
                    "Key": "Name",
                    "Value": "lambda-ap"
                }
            ],
            "AccessPointId": "fsap-015cxmplb72b405fd",
```
Mounting an Amazon EFS file system in another AWS account

You can configure a function to mount an Amazon EFS file system in another AWS account. Before you mount the file system, you must ensure the following:

- **VPC peering** must be configured, and appropriate routes must be added to the route tables in each VPC.
- The security group for the Amazon EFS file system you want to mount must be configured to allow inbound access from the security group associated with your Lambda function.
- Subnets must be created in each VPC with matching Availability Zone (AZ) IDs.
- **DNS Hostnames** must be enabled in both VPCs.

For your Lambda function to access an Amazon EFS file system in another AWS account, that file system must also have a file system policy that grants permission to your function. To learn how to create a file system policy, see Creating file system policies in the Amazon Elastic File System User Guide.

The following shows an example policy that gives Lambda functions in a specified account permission to perform all API actions on a file system.

```json
{
   "Version": "2012-10-17",
   "Id": "efs-lambda-policy",
   "Statement": [
      {
         "Sid": "efs-lambda-statement",
         "Effect": "Allow",
         "Principal": {
            "AWS": "arn:aws:iam::{LAMBDA-ACCOUNT-ID}:root"
           }
      }
   ]
}
```
Note

The example policy shown uses the wildcard character ("*") to grant permissions for Lambda functions in the specified AWS account to perform any API operation on the filesystem. This includes deleting the filesystem. To limit the operations that other AWS accounts can perform on your filesystem, specify the actions you want to allow explicitly. For a list of possible API operations, see Actions, resources, and condition keys for Amazon Elastic File System.

To configure cross-account file system mounting, you use the AWS Command Line Interface (AWS CLI) update-function-configuration operation.

To mount a file system in another AWS account, run the following command. Use your own function name and replace the Amazon Resource Name (ARN) with the ARN of the Amazon EFS access point for the file system you want to mount. LocalMountPath is the path where the function can access the file system, starting with /mnt/. Ensure that the Lambda mount path matches the access point path for the filesystem. For example, if the access point is /efs, the Lambda mount path must be /mnt/efs.

```
aws lambda update-function-configuration --function-name MyFunction \
```

AWS CloudFormation and AWS SAM

You can use AWS CloudFormation and the AWS Serverless Application Model (AWS SAM) to automate the creation of Lambda applications. To enable a file system connection on an AWS SAM AWS::Serverless::Function resource, use the FileSystemConfigs property.
Example template.yml – File system configuration

Transform: AWS::Serverless-2016-10-31
Resources:
  VPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 10.0.0.0/16
  Subnet1:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId:
        Ref: VPC
      CidrBlock: 10.0.1.0/24
      AvailabilityZone: "us-west-2a"
  EfsSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      VpcId:
        Ref: VPC
      GroupDescription: "mnt target sg"
    SecurityGroupIngress:
      - IpProtocol: -1
        CidrIp: "0.0.0.0/0"
  FileSystem:
    Type: AWS::EFS::FileSystem
    Properties:
      PerformanceMode: generalPurpose
  AccessPoint:
    Type: AWS::EFS::AccessPoint
    Properties:
      FileSystemId:
        Ref: FileSystem
      PosixUser:
        Uid: "1001"
        Gid: "1001"
      RootDirectory:
        CreationInfo:
          OwnerGid: "1001"
          OwnerUid: "1001"
          Permissions: "755"
  MountTarget1:
    Type: AWS::EFS::MountTarget
    Properties:
You must add the DependsOn to ensure that the mount targets are fully created before the Lambda runs for the first time.

For the AWS CloudFormation `AWS::Lambda::Function` type, the property name and fields are the same. For more information, see [Using AWS Lambda with AWS CloudFormation](#).

## Sample applications

The GitHub repository for this guide includes a sample application that demonstrates the use of Amazon EFS with a Lambda function.

- `efs-nodejs` – A function that uses an Amazon EFS file system in a Amazon VPC. This sample includes a VPC, file system, mount targets, and access point configured for use with Lambda.
Lambda function aliases

You can create aliases for your Lambda function. A Lambda alias is a pointer to a function version that you can update. The function's users can access the function version using the alias Amazon Resource Name (ARN). When you deploy a new version, you can update the alias to use the new version, or split traffic between two versions.

Sections

- Creating a function alias (Console)
- Managing aliases with the Lambda API
- Managing aliases with AWS SAM and AWS CloudFormation
- Using aliases
- Resource policies
- Alias routing configuration

Creating a function alias (Console)

You can create a function alias using the Lambda console.

To create an alias

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose Aliases and then choose Create alias.
4. On the Create alias page, do the following:
   a. Enter a Name for the alias.
   b. (Optional) Enter a Description for the alias.
   c. For Version, choose a function version that you want the alias to point to.
   d. (Optional) To configure routing on the alias, expand Weighted alias. For more information, see Alias routing configuration.
   e. Choose Save.
Managing aliases with the Lambda API

To create an alias using the AWS Command Line Interface (AWS CLI), use the `create-alias` command.

```bash
aws lambda create-alias --function-name my-function --name alias-name --function-version version-number --description ""
```

To change an alias to point a new version of the function, use the `update-alias` command.

```bash
aws lambda update-alias --function-name my-function --name alias-name --function-version version-number
```

To delete an alias, use the `delete-alias` command.

```bash
aws lambda delete-alias --function-name my-function --name alias-name
```

The AWS CLI commands in the preceding steps correspond to the following Lambda API operations:

- CreateAlias
- UpdateAlias
- DeleteAlias

Managing aliases with AWS SAM and AWS CloudFormation

You can create and manage function aliases using the AWS Serverless Application Model (AWS SAM) and AWS CloudFormation.

To see how to declare a function alias in an AWS SAM template, refer to the `AWS::Serverless::Function` page in the AWS SAM Developer Guide. For information on creating and configuring aliases using AWS CloudFormation, see `AWS::Lambda::Alias` in the AWS CloudFormation User Guide.

Using aliases

Each alias has a unique ARN. An alias can point only to a function version, not to another alias. You can update an alias to point to a new version of the function.
Event sources such as Amazon Simple Storage Service (Amazon S3) invoke your Lambda function. These event sources maintain a mapping that identifies the function to invoke when events occur. If you specify a Lambda function alias in the mapping configuration, you don't need to update the mapping when the function version changes. For more information, see Lambda event source mappings.

In a resource policy, you can grant permissions for event sources to use your Lambda function. If you specify an alias ARN in the policy, you don't need to update the policy when the function version changes.

**Resource policies**

You can use a resource-based policy to give a service, resource, or account access to your function. The scope of that permission depends on whether you apply it to an alias, a version, or the entire function. For example, if you use an alias name (such as helloworld:PROD), the permission allows you to invoke the helloworld function using the alias ARN (helloworld:PROD).

If you attempt to invoke the function without an alias or a specific version, then you get a permission error. This permission error still occurs even if you attempt to directly invoke the function version associated with the alias.

For example, the following AWS CLI command grants Amazon S3 permissions to invoke the PROD alias of the helloworld function when Amazon S3 is acting on behalf of examplebucket.

```
aws lambda add-permission --function-name helloworld \
--qualifier PROD --statement-id 1 --principal s3.amazonaws.com --action
lambda:InvokeFunction \
--source-arn arn:aws:s3:::examplebucket --source-account 123456789012
```

For more information about using resource names in policies, see Resources and conditions for Lambda actions.

**Alias routing configuration**

Use routing configuration on an alias to send a portion of traffic to a second function version. For example, you can reduce the risk of deploying a new version by configuring the alias to send most of the traffic to the existing version, and only a small percentage of traffic to the new version.

Note that Lambda uses a simple probabilistic model to distribute the traffic between the two function versions. At low traffic levels, you might see a high variance between the configured and
actual percentage of traffic on each version. If your function uses provisioned concurrency, you can avoid spillover invocations by configuring a higher number of provisioned concurrency instances during the time that alias routing is active.

You can point an alias to a maximum of two Lambda function versions. The versions must meet the following criteria:

- Both versions must have the same execution role.
- Both versions must have the same dead-letter queue configuration, or no dead-letter queue configuration.
- Both versions must be published. The alias cannot point to $LATEST.

To configure routing on an alias

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose Aliases and then choose Create alias.
4. On the Create alias page, do the following:
   a. Enter a Name for the alias.
   b. (Optional) Enter a Description for the alias.
   c. For Version, choose the first function version that you want the alias to point to.
   d. Expand Weighted alias.
   e. For Additional version, choose the second function version that you want the alias to point to.
   f. For Weight (%), enter a weight value for the function. Weight is the percentage of traffic that is assigned to that version when the alias is invoked. The first version receives the residual weight. For example, if you specify 10 percent to Additional version, the first version is assigned 90 percent automatically.
   g. Choose Save.
Configuring alias routing using CLI

Use the create-alias and update-alias AWS CLI commands to configure the traffic weights between two function versions. When you create or update the alias, you specify the traffic weight in the routing-config parameter.

The following example creates a Lambda function alias named `routing-alias` that points to version 1 of the function. Version 2 of the function receives 3 percent of the traffic. The remaining 97 percent of traffic is routed to version 1.

```bash
aws lambda create-alias --name routing-alias --function-name my-function --function-version 1  
   --routing-config AdditionalVersionWeights={"2"=0.03}
```

Use the `update-alias` command to increase the percentage of incoming traffic to version 2. In the following example, you increase the traffic to 5 percent.

```bash
aws lambda update-alias --name routing-alias --function-name my-function  
   --routing-config AdditionalVersionWeights={"2"=0.05}
```

To route all traffic to version 2, use the `update-alias` command to change the function-version property to point the alias to version 2. The command also resets the routing configuration.

```bash
aws lambda update-alias --name routing-alias --function-name my-function  
   --function-version 2 --routing-config AdditionalVersionWeights={}
```

The AWS CLI commands in the preceding steps correspond to the following Lambda API operations:

- CreateAlias
- UpdateAlias

Determining which version has been invoked

When you configure traffic weights between two function versions, there are two ways to determine the Lambda function version that has been invoked:
• **CloudWatch Logs** – Lambda automatically emits a START log entry that contains the invoked version ID to Amazon CloudWatch Logs for every function invocation. The following is an example:

```
19:44:37 START RequestId: request id Version: $version
```

For alias invocations, Lambda uses the Executed Version dimension to filter the metric data by the invoked version. For more information, see [Working with Lambda function metrics](#).

• **Response payload (synchronous invocations)** – Responses to synchronous function invocations include an `x-amz-executed-version` header to indicate which function version has been invoked.
Lambda function versions

You can use versions to manage the deployment of your functions. For example, you can publish a new version of a function for beta testing without affecting users of the stable production version. Lambda creates a new version of your function each time that you publish the function. The new version is a copy of the unpublished version of the function. The unpublished version is named $LATEST.

Note

You can't publish your function and create a new version if the unpublished version ($LATEST) is the same as the previous published version. You need to deploy code changes or make updates to your function's environment variables in $LATEST before you can create a new version.

A function version includes the following information:

- The function code and all associated dependencies.
- The Lambda runtime identifier and runtime version used by the function.
- All the function settings, including the environment variables.
- A unique Amazon Resource Name (ARN) to identify the specific version of the function.

When using runtime management controls with Auto mode, the runtime version used by the function version is updated automatically. When using Function update or Manual mode, the runtime version is not updated. For more information, see the section called “Runtime updates”.

Sections

- Creating function versions
- Using versions
- Granting permissions

Creating function versions

You can change the function code and settings only on the unpublished version of a function. When you publish a version, Lambda locks the code and most of the settings to maintain a
consistent experience for users of that version. For more information about configuring function settings, see Configuring Lambda function options.

You can create a function version using the Lambda console.

**To create a new function version**

1. Open the Functions page of the Lambda console.
2. Choose a function and then choose Versions.
3. On the versions configuration page, choose Publish new version.
4. (Optional) Enter a version description.
5. Choose Publish.

Alternatively, you can publish a version of a function using the PublishVersion API operation.

The following AWS CLI command publishes a new version of a function. The response returns configuration information about the new version, including the version number and the function ARN with the version suffix.

```bash
aws lambda publish-version --function-name my-function
```

You should see the following output:

```json
{
   "FunctionName": "my-function",
   "Version": "1",
   "Role": "arn:aws:iam::123456789012:role/lambda-role",
   "Handler": "function.handler",
   "Runtime": "nodejs20.x",
   ...
}
```

**Note**

Lambda assigns monotonically increasing sequence numbers for versioning. Lambda never reuses version numbers, even after you delete and recreate a function.
Using versions

You can reference your Lambda function using either a qualified ARN or an unqualified ARN.

- **Qualified ARN** – The function ARN with a version suffix. The following example refers to version 42 of the helloworld function.


- **Unqualified ARN** – The function ARN without a version suffix.

  arn:aws:lambda:aws-region:acct-id:function:helloworld

You can use a qualified or an unqualified ARN in all relevant API operations. However, you can't use an unqualified ARN to create an alias.

If you decide not to publish function versions, you can invoke the function using either the qualified or unqualified ARN in your event source mapping. When you invoke a function using an unqualified ARN, Lambda implicitly invokes $LATEST.

Lambda publishes a new function version only if the code has never been published or if the code has changed from the last published version. If there is no change, the function version remains at the last published version.

The qualified ARN for each Lambda function version is unique. After you publish a version, you can't change the ARN or the function code.

Granting permissions

You can use a resource-based policy or an identity-based policy to grant access to your function. The scope of the permission depends on whether you apply the policy to a function or to one version of a function. For more information about function resource names in policies, see Resources and conditions for Lambda actions.

You can simplify the management of event sources and AWS Identity and Access Management (IAM) policies by using function aliases. For more information, see Lambda function aliases.
Configuring a Lambda function to stream responses

You can configure your Lambda function URLs to stream response payloads back to clients. Response streaming can benefit latency sensitive applications by improving time to first byte (TTFB) performance. This is because you can send partial responses back to the client as they become available. Additionally, you can use response streaming to build functions that return larger payloads. Response stream payloads have a soft limit of 20 MB as compared to the 6 MB limit for buffered responses. Streaming a response also means that your function doesn’t need to fit the entire response in memory. For very large responses, this can reduce the amount of memory you need to configure for your function.

The speed at which Lambda streams your responses depends on the response size. The streaming rate for the first 6MB of your function’s response is uncapped. For responses larger than 6MB, the remainder of the response is subject to a bandwidth cap. For more information on streaming bandwidth, see Bandwidth limits for response streaming.

Streaming responses incurs a cost. For more information, see AWS Lambda Pricing.

Lambda supports response streaming on Node.js managed runtimes. For other languages, you can use a custom runtime with a custom Runtime API integration to stream responses or use the Lambda Web Adapter. You can stream responses through Lambda Function URLs, the AWS SDK, or using the Lambda InvokeWithResponseStream API.

Note

When testing your function through the Lambda console, you'll always see responses as buffered.

Writing response streaming-enabled functions

Writing the handler for response streaming functions is different than typical handler patterns. When writing streaming functions, be sure to do the following:

• Wrap your function with the awslambda.streamifyResponse() decorator that the native Node.js runtimes provide.
• End the stream gracefully to ensure that all data processing is complete.
Configuring a handler function to stream responses

To indicate to the runtime that Lambda should stream your function's responses, you must wrap your function with the streamifyResponse() decorator. This tells the runtime to use the proper logic path for streaming responses and enables the function to stream responses.

The streamifyResponse() decorator accepts a function that accepts the following parameters:

- `event` – Provides information about the function URL's invocation event, such as the HTTP method, query parameters, and the request body.
- `responseStream` – Provides a writable stream.
- `context` – Provides methods and properties with information about the invocation, function, and execution environment.

The responseStream object is a Node.js writableStream. As with any such stream, you should use the pipeline() method.

Example response streaming-enabled handler

```javascript
const pipeline = require("util").promisify(require("stream").pipeline);
const { Readable } = require('stream');

exports.echo = awslambda.streamifyResponse(async (event, responseStream, _context) => {
    // As an example, convert event to a readable stream.
    const requestStream = Readable.from(Buffer.from(JSON.stringify(event)));

    await pipeline(requestStream, responseStream);
});
```

While responseStream offers the write() method to write to the stream, we recommend that you use pipeline() wherever possible. Using pipeline() ensures that the writable stream is not overwhelmed by a faster readable stream.

Ending the stream

Make sure that you properly end the stream before the handler returns. The pipeline() method handles this automatically.
For other use cases, call the `responseStream.end()` method to properly end a stream. This method signals that no more data should be written to the stream. This method isn't required if you write to the stream with `pipeline()` or `pipe()`.

**Example Example ending a stream with pipeline()**

```javascript
const pipeline = require("util").promisify(require("stream").pipeline);

exports.handler = awslambda.streamifyResponse(async (event, responseStream, _context) => {
  await pipeline(requestStream, responseStream);
});
```

**Example Example ending a stream without pipeline()**

```javascript
exports.handler = awslambda.streamifyResponse(async (event, responseStream, _context) => {
  responseStream.write("Hello ");
  responseStream.write("world ");
  responseStream.write("from ");
  responseStream.write("Lambda!");
  responseStream.end();
});
```

### Invoking a response streaming enabled function using Lambda function URLs

**Note**

You must invoke your function using a function URL to stream the responses.

You can invoke response streaming enabled functions by changing the invoke mode of your function's URL. The invoke mode determines which API operation Lambda uses to invoke your function. The available invoke modes are:

- **BUFFERED** – This is the default option. Lambda invokes your function using the Invoke API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.
**RESPONSE_STREAM** – Enables your function to stream payload results as they become available. Lambda invokes your function using the `InvokeWithResponseStream` API operation. The maximum response payload size is 20 MB. However, you can [request a quota increase](#).

You can still invoke your function without response streaming by directly calling the `Invoke` API operation. However, Lambda streams all response payloads for invocations that come through the function's URL until you change the invoke mode to BUFFERED.

### To set the invoke mode of a function URL (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of the function that you want to set the invoke mode for.
3. Choose the **Configuration** tab, and then choose **Function URL**.
4. Choose **Edit**, then choose **Additional settings**.
5. Under **Invoke mode**, choose your desired invoke mode.
6. Choose **Save**.

### To set the invoke mode of a function's URL (AWS CLI)

```bash
aws lambda update-function-url-config --function-name my-function --invoke-mode RESPONSE_STREAM
```

### To set the invoke mode of a function's URL (AWS CloudFormation)

```yaml
MyFunctionUrl:
  Type: AWS::Lambda::Url
  Properties:
    AuthType: AWS_IAM
    InvokeMode: RESPONSE_STREAM
```

For more information about configuring function URLs, see [Lambda function URLs](#).

### Bandwidth limits for response streaming

The first 6MB of your function’s response payload has uncapped bandwidth. After this initial burst, Lambda streams your response at a maximum rate of 2MBps. If your function responses never exceed 6MB, then this bandwidth limit never applies.
### Note

Bandwidth limits only apply to your function’s response payload, and not to network access by your function.

The rate of uncapped bandwidth varies depending on a number of factors, including your function's processing speed. You can normally expect a rate higher than 2MBps for the first 6MB of your function’s response. If your function is streaming a response to a destination outside of AWS, the streaming rate also depends on the speed of the external internet connection.

### Tutorial: Creating a response streaming Lambda function with a function URL

In this tutorial, you create a Lambda function defined as a .zip file archive with a function URL endpoint that returns a response stream. For more information about configuring function URLs, see [Creating and managing function URLs](#).

### Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven’t already, follow the instructions in [Create a Lambda function with the console](#) to create your first Lambda function.

To complete the following steps, you need the [AWS Command Line Interface (AWS CLI) version 2](#). Commands and the expected output are listed in separate blocks:

```bash
aws --version
```

You should see the following output:

```bash
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.
Create an execution role

Create the execution role that gives your Lambda function permission to access AWS resources.

To create an execution role

1. Open the Roles page of the AWS Identity and Access Management (IAM) console.
2. Choose Create role.
3. Create a role with the following properties:
   - Trusted entity type – AWS service
   - Use case – Lambda
   - Permissions – AWSLambdaBasicExecutionRole
   - Role name – response-streaming-role

The AWSLambdaBasicExecutionRole policy has the permissions that the function needs to write logs to Amazon CloudWatch Logs. After you create the role, note down the its Amazon Resource Name (ARN). You'll need it in the next step.

Create a response streaming function (AWS CLI)

Create a response streaming Lambda function with a function URL endpoint using the AWS Command Line Interface (AWS CLI).

To create a function that can stream responses

1. Copy the following code example into a file named index.mjs.

```javascript
import util from 'util';
```
import stream from 'stream';
const { Readable } = stream;
const pipeline = util.promisify(stream.pipeline);

/* global awslambda */
export const handler = awslambda.streamifyResponse(async (event, responseStream, _context) => {
  const requestStream = Readable.from(Buffer.from(JSON.stringify(event)));
  await pipeline(requestStream, responseStream);
});

2. Create a deployment package.

```
zip function.zip index.mjs
```

3. Create a Lambda function with the `create-function` command. Replace the value of `--role` with the role ARN from the previous step.

```
aws lambda create-function \
  --function-name my-streaming-function \
  --runtime nodejs16.x \
  --zip-file fileb://function.zip \
  --handler index.handler \
  --role arn:aws:iam::123456789012:role/response-streaming-role
```

To create a function URL

1. Add a resource-based policy to your function to allow access to your function URL. Replace the value of `--principal` with your AWS account ID.

```
aws lambda add-permission \
  --function-name my-streaming-function \
  --action lambda:InvokeFunctionUrl \
  --statement-id 12345 \
  --principal 123456789012 \
  --function-url-auth-type AWS_IAM \
  --statement-id url
```

2. Create a URL endpoint for the function with the `create-function-url-config` command.

```
aws lambda create-function-url-config
```
Create a response streaming function (AWS CloudFormation)

You can also create a streaming function configured with a function URL using AWS CloudFormation.

<table>
<thead>
<tr>
<th>Resources:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyStreamingFunction:</td>
</tr>
<tr>
<td>Type: AWS::Lambda::Function</td>
</tr>
<tr>
<td>Properties:</td>
</tr>
<tr>
<td>Handler: index.handler</td>
</tr>
<tr>
<td>Runtime: nodejs16.x</td>
</tr>
<tr>
<td>Role: <code>arn:aws:iam::123456789012:role/response-streaming-role</code></td>
</tr>
<tr>
<td>Code:</td>
</tr>
<tr>
<td>ZipFile:</td>
</tr>
</tbody>
</table>
| `exports.handler = awslambda.streamifyResponse(
async (event, responseStream, _context) => {
  // Metadata is a JSON serializable JS object. Its shape is not defined here.
  const metadata = {
    statusCode: 200,
    headers: {
      "Content-Type": "application/json",
      "CustomHeader": "outerspace"
    }
  };

  // Assign to the responseStream parameter to prevent accidental reuse of the non-wrapped stream.
  responseStream = awslambda.HttpResponseStream.from(responseStream, metadata);

  responseStream.write("Streaming with Helper \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 0 \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 1 \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 3 \n");` |
await new Promise(r => setTimeout(r, 1000));
responseStream.end();
await responseStream.finished();
}

Description: Create a streaming function configured with a URL.

MyStreamingFunctionPermissions:
Type: AWS::Lambda::Permission
Properties:
  FunctionName: !Ref MyStreamingFunction
  Action: lambda:InvokeFunctionUrl
  Principal: 123456789012
  FunctionUrlAuthType: AWS_IAM

MyStreamingFunctionUrl:
Type: AWS::Lambda::Url
Properties:
  TargetFunctionArn: !Ref MyUrlFunction
  AuthType: AWS_IAM
  InvokeMode: RESPONSE_STREAM

Create a response streaming function (AWS SAM)

You can also create a streaming function configured with a function URL using AWS Serverless Application Model (AWS SAM).

AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  StreamingFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: "<path-to-your-code>"
      Handler: index.handler
      Runtime: nodejs16.x
      AutoPublishAlias: live
  MyStreamingFunctionUrl:
    Type: AWS::Lambda::Url
    Properties:
      TargetFunctionArn: !Ref StreamingFunction
      AuthType: AWS_IAM
      InvokeMode: RESPONSE_STREAM
Test the function URL endpoint

Test your integration by invoking your function. You can open your function's URL in a browser, or you can use curl.

```bash
curl --request GET "<function_url>" --user "<key:token>" --aws-sigv4 "aws:amz:us-east-1:lambda" --no-buffer
```

Our function URL uses the IAM_AUTH authentication type. This means that you need to sign requests with both your AWS access key and secret key. In the previous command, replace `<key:token>` with the AWS access key ID. Enter your AWS secret key when prompted. If you don't have your AWS secret key, you can use temporary AWS credentials instead.

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose Delete.
4. Enter the name of the role in the text input field and choose Delete.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose Actions, Delete.
4. Type delete in the text input field and choose Delete.
Deploying Lambda functions

You can deploy code to your Lambda function by uploading a zip file archive, or by creating and uploading a container image.

.zip file archives

A .zip file archive includes your application code and its dependencies. When you author functions using the Lambda console or a toolkit, Lambda automatically creates a .zip file archive of your code.

When you create functions with the Lambda API, command line tools, or the AWS SDKs, you must create a deployment package. You also must create a deployment package if your function uses a compiled language, or to add dependencies to your function. To deploy your function's code, you upload the deployment package from Amazon Simple Storage Service (Amazon S3) or your local machine.

You can upload a .zip file as your deployment package using the Lambda console, AWS Command Line Interface (AWS CLI), or to an Amazon Simple Storage Service (Amazon S3) bucket.

Container images

You can package your code and dependencies as a container image using tools such as the Docker command line interface (CLI). You can then upload the image to your container registry hosted on Amazon Elastic Container Registry (Amazon ECR).

AWS provides a set of open-source base images that you can use to build the container image for your function code. You can also use alternative base images from other container registries. AWS also provides an open-source runtime client that you add to your alternative base image to make it compatible with the Lambda service.

Additionally, AWS provides a runtime interface emulator for you to test your functions locally using tools such as the Docker CLI.

Note

You create each container image to be compatible with one of the instruction set architectures that Lambda supports. Lambda provides base images for each of the
instruction set architectures and Lambda also provides base images that support both architectures. The image that you build for your function must target only one of the architectures.

There is no additional charge for packaging and deploying functions as container images. When a function deployed as a container image is invoked, you pay for invocation requests and execution duration. You do incur charges related to storing your container images in Amazon ECR. For more information, see Amazon ECR pricing.

Topics

- Deploying Lambda functions as .zip file archives
- Working with Lambda container images
- Testing Lambda container images locally
Deploying Lambda functions as .zip file archives

When you create a Lambda function, you package your function code into a deployment package. Lambda supports two types of deployment packages: container images and .zip file archives. The workflow to create a function depends on the deployment package type. To configure a function defined as a container image, see the section called “Container images”.

You can use the Lambda console and the Lambda API to create a function defined with a .zip file archive. You can also upload an updated .zip file to change the function code.

**Note**

You cannot convert an existing container image function to use a .zip file archive. You must create a new function.

Topics

- Creating the function
- Using the console code editor
- Updating function code
- Changing the runtime
- Changing the architecture
- Using the Lambda API
- AWS CloudFormation

Creating the function

When you create a function defined with a .zip file archive, you choose a code template, the language version, and the execution role for the function. You add your function code after Lambda creates the function.

**To create the function**

1. Open the Functions page of the Lambda console.
2. Choose Create function.
3. Choose Author from scratch or Use a blueprint to create your function.

4. Under Basic information, do the following:

   a. For Function name, enter the function name. Function names are limited to 64 characters in length.
   b. For Runtime, choose the language version to use for your function.
   c. (Optional) For Architecture, choose the instruction set architecture to use for your function. The default architecture is x86_64. When you build the deployment package for your function, make sure that it is compatible with this instruction set architecture.

5. (Optional) Under Permissions, expand Change default execution role. You can create a new Execution role or use an existing role.

6. (Optional) Expand Advanced settings. You can choose a Code signing configuration for the function. You can also configure an (Amazon VPC) for the function to access.

7. Choose Create function.

Lambda creates the new function. You can now use the console to add the function code and configure other function parameters and features. For code deployment instructions, see the handler page for the runtime your function uses.

Node.js

   Deploy Node.js Lambda functions with .zip file archives

Python

   Working with .zip file archives for Python Lambda functions

Ruby

   Working with .zip file archives for Ruby Lambda functions

Java

   Deploy Java Lambda functions with .zip or JAR file archives

Go

   Deploy Go Lambda functions with .zip file archives

C# 

   Build and deploy C# Lambda functions with .zip file archives
Deploy PowerShell Lambda functions with .zip file archives

Using the console code editor

The console creates a Lambda function with a single source file. For scripting languages, you can edit this file and add more files using the built-in code editor. To save your changes, choose Save. Then, to run your code, choose Test.

Note

The Lambda console uses AWS Cloud9 to provide an integrated development environment in the browser. You can also use AWS Cloud9 to develop Lambda functions in your own environment. For more information, see Working with AWS Lambda functions using the AWS Toolkit in the AWS Cloud9 user guide.

When you save your function code, the Lambda console creates a .zip file archive deployment package. When you develop your function code outside of the console (using an IDE) you need to create a deployment package to upload your code to the Lambda function.

Updating function code

For scripting languages (Node.js, Python, and Ruby), you can edit your function code in the embedded code editor. If the code is larger than 3MB, or if you need to add libraries, or for languages that the editor doesn't support (Java, Go, C#), you must upload your function code as a .zip archive. If the .zip file archive is smaller than 50 MB, you can upload the .zip file archive from your local machine. If the file is larger than 50 MB, upload the file to the function from an Amazon S3 bucket.

To upload function code as a .zip archive

1. Open the Functions page of the Lambda console.
2. Choose the function to update and choose the Code tab.
4. Choose .zip file, and then choose Upload.
In the file chooser, select the new image version, choose **Open**, and then choose **Save**.

5. (Alternative to step 4) Choose **Amazon S3 location**.
   - In the text box, enter the S3 link URL of the .zip file archive, then choose **Save**.

### Changing the runtime

If you update the function configuration to use a new runtime, you may need to update the function code to be compatible with the new runtime. If you update the function configuration to use a different runtime, you **must** provide new function code that is compatible with the runtime and architecture. For instructions on how to create a deployment package for the function code, see the handler page for the runtime that the function uses.

**To change the runtime**

1. Open the **Functions page** of the Lambda console.
2. Choose the function to update and choose the **Code** tab.
3. Scroll down to the **Runtime settings** section, which is under the code editor.
4. Choose **Edit**.
   
   a. For **Runtime**, select the runtime identifier.
   
   b. For **Handler**, specify file name and handler for your function.
   
   c. For **Architecture**, choose the instruction set architecture to use for your function.
5. Choose **Save**.

### Changing the architecture

Before you can change the instruction set architecture, you need to ensure that your function's code is compatible with the target architecture.

If you use Node.js, Python, or Ruby and you edit your function code in the embedded **editor**, the existing code may run without modification.

However, if you provide your function code using a .zip file archive deployment package, you must prepare a new .zip file archive that is compiled and built correctly for the target runtime and instruction-set architecture. For instructions, see the handler page for your function runtime.
To change the instruction set architecture

1. Open the Functions page of the Lambda console.
2. Choose the function to update and choose the Code tab.
4. For Architecture, choose the instruction set architecture to use for your function.
5. Choose Save.

Using the Lambda API

To create and configure a function that uses a .zip file archive, use the following API operations:

- CreateFunction
- UpdateFunctionCode
- UpdateFunctionConfiguration

AWS CloudFormation

You can use AWS CloudFormation to create a Lambda function that uses a .zip file archive. In your AWS CloudFormation template, the AWS::Lambda::Function resource specifies the Lambda function. For descriptions of the properties in the AWS::Lambda::Function resource, see AWS::Lambda::Function in the AWS CloudFormation User Guide.

In the AWS::Lambda::Function resource, set the following properties to create a function defined as a .zip file archive:

- AWS::Lambda::Function
  - PackageType – Set to Zip.
  - Code – Enter the Amazon S3 bucket name and .zip file name in the S3Bucket and S3Keyfields. For Node.js or Python, you can provide inline source code of your Lambda function.
  - Runtime – Set the runtime value.
  - Architecture – Set the architecture value to arm64 to use the AWS Graviton2 processor. By default, the architecture value is x86_64.
Working with Lambda container images

Your AWS Lambda function’s code consists of scripts or compiled programs and their dependencies. You use a deployment package to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives.

There are three ways to build a container image for a Lambda function:

- **Using an AWS base image for Lambda**

  The AWS base images are preloaded with a language runtime, a runtime interface client to manage the interaction between Lambda and your function code, and a runtime interface emulator for local testing.

- **Using an AWS OS-only base image**

  AWS OS-only base images contain an Amazon Linux distribution and the runtime interface emulator. These images are commonly used to create container images for compiled languages, such as Go and Rust, and for a language or language version that Lambda doesn’t provide a base image for, such as Node.js 19. You can also use OS-only base images to implement a custom runtime. To make the image compatible with Lambda, you must include a runtime interface client for your language in the image.

- **Using a non-AWS base image**

  You can use an alternative base image from another container registry, such as Alpine Linux or Debian. You can also use a custom image created by your organization. To make the image compatible with Lambda, you must include a runtime interface client for your language in the image.

**Tip**
To reduce the time it takes for Lambda container functions to become active, see Use multi-stage builds in the Docker documentation. To build efficient container images, follow the Best practices for writing Dockerfiles.

To create a Lambda function from a container image, build your image locally and upload it to an Amazon Elastic Container Registry (Amazon ECR) repository. Then, specify the repository URI when you create the function. The Amazon ECR repository must be in the same AWS Region as the
Lambda function. You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

This page explains the base image types and requirements for creating Lambda-compatible container images.

Topics

- Requirements
- Using an AWS base image for Lambda
- Using an AWS OS-only base image
- Using a non-AWS base image
- Runtime interface clients
- Amazon ECR permissions
- Container image settings

Requirements

Install the AWS Command Line Interface (AWS CLI) version 2 and the Docker CLI. Additionally, note the following requirements:

- The container image must implement the Lambda runtime API. The AWS open-source runtime interface clients implement the API. You can add a runtime interface client to your preferred base image to make it compatible with Lambda.
- The container image must be able to run on a read-only file system. Your function code can access a writable /tmp directory with between 512 MB and 10,240 MB, in 1-MB increments, of storage.
- The default Lambda user must be able to read all the files required to run your function code. Lambda follows security best practices by defining a default Linux user with least-privileged permissions. Verify that your application code does not rely on files that other Linux users are restricted from running.
- Lambda supports only Linux-based container images.
- Lambda provides multi-architecture base images. However, the image you build for your function must target only one of the architectures. Lambda does not support functions that use multi-architecture container images.
Using an AWS base image for Lambda

You can use one of the [AWS base images](#) for Lambda to build the container image for your function code. The base images are preloaded with a language runtime and other components required to run a container image on Lambda. You add your function code and dependencies to the base image and then package it as a container image.

AWS periodically provides updates to the AWS base images for Lambda. If your Dockerfile includes the image name in the FROM property, your Docker client pulls the latest version of the image from the [Amazon ECR repository](#). To use the updated base image, you must rebuild your container image and [update the function code](#).

To build a container image using an AWS base image, choose the instructions for your preferred language:

- [Node.js](#)
- [TypeScript](#) (uses a Node.js base image)
- [Python](#)
- [Java](#)
- [Go](#)
- [.NET](#)
- [Ruby](#)

Using an AWS OS-only base image

[AWS OS-only base images](#) contain an Amazon Linux distribution and the runtime interface emulator. These images are commonly used to create container images for compiled languages, such as Go and Rust, and for a language or language version that Lambda doesn't provide a base image for, such as Node.js 19. You can also use OS-only base images to implement a custom runtime. To make the image compatible with Lambda, you must include a runtime interface client for your language in the image.
Amazon Elastic Container Registry Public Gallery: gallery.ecr.aws/lambda/provided

The Amazon Linux 2023 (provided.al2023) base image provides several advantages over Amazon Linux 2. Because the provided.al2023 base image is based on the AL2023 minimal container image, it is less than 40 MB—compared to more than 100 MB for the provided.al2 base image. The provided.al2023 base image also provides updated versions of libraries such as glibc.

The provided.al2023 base image uses microdnf (symlinked as dnf) as the package manager instead of yum, which is the default package manager in Amazon Linux 2. microdnf is a standalone implementation of dnf. For a list of packages that are included in the provided.al2023 base image, refer to the Minimal Container columns in Comparing packages installed on Amazon Linux 2023 Container Images. For more information about the differences between provided.al2023 and provided.al2, see Introducing the Amazon Linux 2023 runtime for AWS Lambda on the AWS Compute Blog.

### Using a non-AWS base image

Lambda supports any image that conforms to one of the following image manifest formats:

- Docker image manifest V2, schema 2 (used with Docker version 1.10 and newer)
- Open Container Initiative (OCI) Specifications (v1.0.0 and up)

Lambda supports a maximum uncompressed image size of 10 GB, including all layers.
Note
To make the image compatible with Lambda, you must include a runtime interface client for your language in the image.

Runtime interface clients
If you use an OS-only base image or an alternative base image, you must include a runtime interface client in your image. The runtime interface client must extend the Lambda runtime API, which manages the interaction between Lambda and your function code. AWS provides open-source runtime interface clients for the following languages:

- **Node.js**
- **Python**
- **Java**
- **.NET**
- **Go**
- **Ruby**
- **Rust** – The Rust runtime client is an experimental package. It is subject to change and intended only for evaluation purposes.

If you’re using a language that doesn't have an AWS-provided runtime interface client, you must create your own.

Amazon ECR permissions
Before you create a Lambda function from a container image, you must build the image locally and upload it to an Amazon ECR repository. When you create the function, specify the Amazon ECR repository URI.

Make sure that the permissions for the user or role that creates the function contain the AWS managed policies GetRepositoryPolicy and SetRepositoryPolicy.

For example, use the IAM console to create a role with the following policy:

```json
{
}
```
Amazon ECR repository policies

For a function in the same account as the container image in Amazon ECR, you can add `ecr:BatchGetImage` and `ecr:GetDownloadUrlForLayer` permissions to your Amazon ECR repository policy. The following example shows the minimum policy:

```json
{
   "Sid": "LambdaECRImageRetrievalPolicy",
   "Effect": "Allow",
   "Principal": {
      "Service": "lambda.amazonaws.com"
   },
   "Action": [
      "ecr:BatchGetImage",
      "ecr:GetDownloadUrlForLayer"
   ]
}
```

For more information about Amazon ECR repository permissions, see [Private repository policies](#) in the [Amazon Elastic Container Registry User Guide](#).

If the Amazon ECR repository does not include these permissions, Lambda adds `ecr:BatchGetImage` and `ecr:GetDownloadUrlForLayer` to the container image repository permissions. Lambda can add these permissions only if the principal calling Lambda has `ecr:getRepositoryPolicy` and `ecr:setRepositoryPolicy` permissions.

To view or edit your Amazon ECR repository permissions, follow the directions in [Setting a private repository policy statement](#) in the [Amazon Elastic Container Registry User Guide](#).
Amazon ECR cross-account permissions

A different account in the same region can create a function that uses a container image owned by your account. In the following example, your Amazon ECR repository permissions policy needs the following statements to grant access to account number 123456789012.

- **CrossAccountPermission** – Allows account 123456789012 to create and update Lambda functions that use images from this ECR repository.

- **LambdaECRImageCrossAccountRetrievalPolicy** – Lambda will eventually set a function's state to inactive if it is not invoked for an extended period. This statement is required so that Lambda can retrieve the container image for optimization and caching on behalf of the function owned by 123456789012.

Example — Add cross-account permission to your repository

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "CrossAccountPermission",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:root"
      }
    },
    {
      "Sid": "LambdaECRImageCrossAccountRetrievalPolicy",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Condition": {
        "StringLike": {
          "AWS::Region": "us-east-1"
        }
      }
    }
  ]
}
```
To give access to multiple accounts, you add the account IDs to the Principal list in the CrossAccountPermission policy and to the Condition evaluation list in the LambdaECRIImageCrossAccountRetrievalPolicy.

If you are working with multiple accounts in an AWS Organization, we recommend that you enumerate each account ID in the ECR permissions policy. This approach aligns with the AWS security best practice of setting narrow permissions in IAM policies.

**Container image settings**

The following are common container image settings. If you use these settings in your Dockerfile, note how Lambda interprets and processes these settings:

- **ENTRYPOINT** – Specifies the absolute path to the entry point of the application.
- **CMD** – Specifies parameters that you want to pass in with ENTRYPOINT.
- **WORKDIR** – Specifies the absolute path to the working directory.
- **ENV** – Specifies an environment variable for the Lambda function.

For more information about how Docker uses the container image settings, see [ENTRYPOINT](https://docs.docker.com/engine/reference/builder/#entrypoint) in the Dockerfile reference on the Docker Docs website. For more information about using ENTRYPOINT and CMD, see [Demystifying ENTRYPOINT and CMD in Docker](https://aws.amazon.com/blogs/aws/demystifying-entrypoint-and-cmd-in-docker) on the AWS Open Source Blog.

You can specify the container image settings in the Dockerfile when you build your image. You can also override these configurations using the Lambda console or Lambda API. This allows you to deploy multiple functions that deploy the same container image but with different runtime configurations.

⚠️ **Warning**

When you specify ENTRYPOINT or CMD in the Dockerfile or as an override, make sure that you enter the absolute path. Also, do not use symlinks as the entry point to the container.
To override the configuration values in the container image

1. Open the Functions page of the Lambda console.
2. Choose the function to update.
4. Enter new values for any of the override settings, and then choose Save.
5. (Optional) To add or override environment variables, under Environment variables, choose Edit.

For more information, see the section called “Environment variables”.
Testing Lambda container images locally

You can use the Lambda runtime interface emulator to locally test a container image function before uploading it to Amazon Elastic Container Registry (Amazon ECR) and deploying it to Lambda. The emulator is a proxy for the Lambda runtime API. It’s a lightweight web server that converts HTTP requests into JSON events to pass to the Lambda function in the container image.

The AWS base images and OS-only base images include the runtime interface emulator. If you use an alternative base image, such as an Alpine Linux or Debian image, you can build the emulator into your image or install it on your local machine.

The runtime interface emulator is available on the AWS GitHub repository. There are separate packages for the x86-64 and arm64 architectures.

Topics

- Guidelines for using the runtime interface emulator
- Environment variables
- Testing images built from AWS base images
- Testing images built from AWS OS-only base images
- Testing images built from alternative base images

Guidelines for using the runtime interface emulator

Note the following guidelines when using the runtime interface emulator:

- The RIE does not emulate Lambda security and authentication configurations, or Lambda orchestration.
- Lambda provides an emulator for each of the instruction set architectures.
- The emulator does not support AWS X-Ray tracing or other Lambda integrations.

Environment variables

The runtime interface emulator supports a subset of environment variables for the Lambda function in the local running image.

If your function uses security credentials, you can configure the credentials by setting the following environment variables:
AWS ACCESS_KEY_ID
AWS_SECRET_ACCESS_KEY
AWS_SESSION_TOKEN
AWS_DEFAULT_REGION

To set the function timeout, configure AWS_LAMBDA_FUNCTION_TIMEOUT. Enter the maximum number of seconds that you want to allow the function to run.

The emulator does not populate the following Lambda environment variables. However, you can set them to match the values that you expect when the function runs in the Lambda service:

- AWS_LAMBDA_FUNCTION_VERSION
- AWS_LAMBDA_FUNCTION_NAME
- AWS_LAMBDA_FUNCTION_MEMORY_SIZE

Testing images built from AWS base images

The AWS base images for Lambda include the runtime interface emulator. After building your Docker image, follow these steps to test it locally.

1. Start the Docker image with the `docker run` command. In this example, `docker-image` is the image name and `test` is the tag.

   ```bash
   docker run --platform linux/amd64 -p 9000:8080 docker-image:tag
   ```

   This command runs the image as a container and creates a local endpoint at localhost:9000/2015-03-31/functions/function/invocations.

   **Note**

   If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

2. From a new terminal window, post an event to the local endpoint.
Linux/macOS

In Linux and macOS, run the following `curl` command:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{"payload":"hello world!"}'
```

PowerShell

In PowerShell, run the following `Invoke-WebRequest` command:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{}' -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType "application/json"
```

3. Get the container ID.

```bash
docker ps
```

4. Use the `docker kill` command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```bash
docker kill 3766c4ab331c
```
Testing images built from AWS OS-only base images

The **AWS OS-only base images** include the runtime interface emulator. After building your Docker image, follow these steps to test it locally.

1. Start the Docker image with the `docker run` command.

   **Node.js**

   ```bash
   docker run -d -v ~/.aws-lambda-rie:/aws-lambda -p 9000:8080 \
   --entrypoint /aws-lambda/aws-lambda-rie \
   docker-image:test \
   /usr/local/bin/npx aws-lambda-ric index.handler
   ```

   - `/usr/local/bin/npx aws-lambda-ric`: The `npx` command to start the Node.js runtime interface client
   - `index.handler`: The Lambda function handler (usually the CMD in your Dockerfile)

   **Python**

   ```bash
   docker run -d -v ~/.aws-lambda-rie:/aws-lambda -p 9000:8080 \
   --entrypoint /aws-lambda/aws-lambda-rie \
   docker-image:test \
   /usr/local/bin/python -m awslambdaric app.handler
   ```

   - `/usr/local/bin/python -m awslambdaric`: The Python interpreter command to run the Python runtime interface client as a script
   - `app.handler`: The Lambda function handler (usually the CMD in your Dockerfile)

   **Java**

   ```bash
   docker run -d -v ~/.aws-lambda-rie:/aws-lambda -p 9000:8080 \
   --entrypoint /aws-lambda/aws-lambda-rie \
   docker-image:test \
   /usr/bin/java -cp './' \
   com.amazonaws.services.lambda.runtime.api.client.AWSLambda
   example.App::sayHello
   ```
• /usr/bin/java -cp './/*'
  com.amazonaws.services.lambda.runtime.api.client.AWSLambda: Sets the classpath to the Java runtime interface client

• app.handlerexample.App::sayHello: The Lambda function handler (usually the CMD in your Dockerfile)

Go

In the following example, /main is the binary that is compiled during the Docker build.

```
docker run -d -v ~/.aws-lambda-rie:/aws-lambda -p 9000:8080 
  --entrypoint /aws-lambda/aws-lambda-rie 
  docker-image:test 
  /main
```

Ruby

```
docker run -d -v ~/.aws-lambda-rie:/aws-lambda -p 9000:8080 
  --entrypoint /aws-lambda/aws-lambda-rie 
  docker-image:test 
  aws_lambda_ric index.LambdaFunction::Handler.process
```

• aws_lambda_ric: The Ruby runtime interface client

• index.LambdaFunction::Handler.process: The Lambda function handler (usually the CMD in your Dockerfile)

This command runs the image as a container and creates a local endpoint at localhost:9000/2015-03-31/functions/function/invocations.

2. Post an event to the following endpoint using a curl command:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function with an empty event and returns a response. Some functions might require a JSON payload. Example:
3. Get the container ID.

docker ps

4. Use the `docker kill` command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

docker kill 3766c4ab331c

Testing images built from alternative base images

If you use an alternative base image, such as an Alpine Linux or Debian image, you can build the emulator into your image or install it on your local machine.

Building the runtime interface emulator into an image

To build the emulator into your image

1. Create a script and save it in your project directory. Set execution permissions for the script file.

   The script checks for the presence of the `AWS_LAMBDA_RUNTIME_API` environment variable, which indicates the presence of the runtime API. If the runtime API is present, the script runs the runtime interface client. Otherwise, the script runs the runtime interface emulator.

   Choose your language to see an example script:

   Node.js

   In the following example, `/usr/local/bin/npx aws-lambda-ric` is the `npx` command to start the Node.js runtime interface client.

   Example entry_script.sh

   ```bash
   #!/bin/sh
   if [ -z "${AWS_LAMBDA_RUNTIME_API}" ]; then
   ```
exec /usr/local/bin/aws-lambda-rie /usr/local/bin/npx aws-lambda-ric $@
else
    exec /usr/local/bin/npx aws-lambda-ric $@
fi

**Note**

If you're using Windows, make sure to save the script with LF line endings. If the script uses CRLF, you'll get an error like this when you try to run the Docker image:

```
exec /entry_script.sh: no such file or directory
```

Python

In the following example, `/usr/local/bin/python -m awslambdaric` is the Python interpreter command to run the Python runtime interface client as a script.

**Example entry_script.sh**

```
#!/bin/sh
if [ -z "${AWS_LAMBDA_RUNTIME_API}" ]; then
    exec /usr/local/bin/aws-lambda-rie /usr/local/bin/python -m awslambdaric $@
else
    exec /usr/local/bin/python -m awslambdaric $@
fi
```

**Note**

If you're using Windows, make sure to save the script with LF line endings. If the script uses CRLF, you'll get an error like this when you try to run the Docker image:

```
exec /entry_script.sh: no such file or directory
```
Java

In the following example, `/usr/bin/java -cp './*' com.amazonaws.services.lambda.runtime.api.client.AWSLambda` sets the classpath to the Java runtime interface client.

Example entry_script.sh

```sh
#!/bin/sh
if [ -z "${AWS_LAMBDA_RUNTIME_API}" ]; then
    exec /usr/local/bin/aws-lambda-rie /usr/bin/java -cp './*' com.amazonaws.services.lambda.runtime.api.client.AWSLambda @
else
    exec /usr/bin/java -cp './*' com.amazonaws.services.lambda.runtime.api.client.AWSLambda @
fi
```

Note

If you're using Windows, make sure to save the script with LF line endings. If the script uses CRLF, you'll get an error like this when you try to run the Docker image:

`exec /entry_script.sh: no such file or directory`

Go

In the following example, `/main` is the binary that is compiled during the Docker build.

Example entry_script.sh

```sh
#!/bin/sh
if [ -z "${AWS_LAMBDA_RUNTIME_API}" ]; then
    exec /usr/local/bin/aws-lambda-rie /main @
else
    exec /main @
fi
```
Note

If you're using Windows, make sure to save the script with LF line endings. If the script uses CRLF, you'll get an error like this when you try to run the Docker image:

```
exec /entry_script.sh: no such file or directory
```

Ruby

In the following example, `aws_lambda_ric` is the Ruby runtime interface client.

Example entry_script.sh

```sh
#!/bin/sh
if [ -z "${AWS_LAMBDA_RUNTIME_API}" ]; then
    exec /usr/local/bin/aws-lambda-rie aws_lambda_ric $@
else
    exec aws_lambda_ric $@
fi
```

Note

If you're using Windows, make sure to save the script with LF line endings. If the script uses CRLF, you'll get an error like this when you try to run the Docker image:

```
exec /entry_script.sh: no such file or directory
```

2. Download the runtime interface emulator for your target architecture from GitHub into your project directory. Lambda provides an emulator for each of the instruction set architectures.

Linux/macOS

```sh
curl -Lo aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie \
    && chmod +x aws-lambda-rie
```
To install the arm64 emulator, replace the GitHub repository URL in the previous command with the following:

https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64

PowerShell


To install the arm64 emulator, replace the Uri with the following:

https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64

3. Add the following lines to your Dockerfile. The ENTRYPOINT includes the script that you created in step 1 and your function handler.

Example lines to add to Dockerfile

In the following example, replace lambda_function.handler with your function handler.

COPY ./entry_script.sh /entry_script.sh
RUN chmod +x /entry_script.sh
ADD aws-lambda-rie /usr/local/bin/aws-lambda-rie
ENTRYPOINT ["/entry_script.sh","lambda_function.handler"]

4. Build the Docker image with the docker build command. The following example names the image docker-image and gives it the test tag.

docker build --platform linux/amd64 -t docker-image:test .

Note
The command specifies the --platform linux/amd64 option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using
the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

5. Start the Docker image with the `docker run` command. In this example, `docker-image` is the image name and `test` is the tag.

   ```bash
   docker run --platform linux/amd64 -p 9000:8080 docker-image:tag
   ``

   This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

   **Note**
   
   If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

6. From a new terminal window, post an event to the local endpoint.

   **Linux/macOS**

   In Linux and macOS, run the following `curl` command:

   ```bash
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ``

   This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

   ```bash
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{"payload":"hello world!"}'
   ``

   **PowerShell**

   In PowerShell, run the following `Invoke-WebRequest` command:

   ```powershell
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```
This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType "application/json"
```

7. Get the container ID.

```
docker ps
```

8. Use the `docker kill` command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

**Install the runtime interface emulator locally**

To install the runtime interface emulator on your local machine, download the package for your preferred architecture from GitHub. Then, use the `docker run` command to start the container image and set the `--entrypoint` to the emulator. For more information, choose the instructions for your preferred language:

- [Node.js](#)
- [Python](#)
- [Java](#)
- [Go](#)
- [Ruby](#)
Invoking Lambda functions

You can invoke Lambda functions directly using the Lambda console, a function URL HTTP(S) endpoint, the Lambda API, an AWS SDK, the AWS Command Line Interface (AWS CLI), and AWS toolkits. You can also configure other AWS services to invoke your function in response to events or external requests, or on a schedule. For example, Amazon Simple Storage Service (Amazon S3) can invoke your function when an object is created in an S3 bucket, or Amazon EventBridge (CloudWatch Events) can invoke your function on a schedule. You can also configure Lambda to read items from a stream or a queue and invoke your function to process them.

When you invoke a function, you can choose to invoke it synchronously or asynchronously. With synchronous invocation, you wait for the function to process the event and return a response. With asynchronous invocation, Lambda queues the event for processing and returns a response immediately. For asynchronous invocation, Lambda handles retries and can send invocation records to a destination.

For another AWS service to invoke your function directly, you need to create a trigger using the Lambda console. A trigger is a resource you configure to allow another AWS service to invoke your function when certain events or conditions occur. Your function can have multiple triggers. Each trigger acts as a client invoking your function independently, and each event that Lambda passes to your function has data from only one trigger.

To create a trigger, open the functions page of the Lambda console and choose the function you want to add a trigger to. In the Function overview pane, choose add trigger, select the AWS service you want to invoke your function, and follow the instructions to create a trigger.

Each service varies in the options you configure for the trigger and in the structure of the event that the service sends to your Lambda function to invoke it. For a full list of the AWS services that can invoke your Lambda function by using a trigger, and for more information about configuring triggers for different services, see Using Lambda with other services.

For your Lambda function to process items from a stream or a queue, such as an Amazon Kinesis stream or an Amazon Simple Queue Service (Amazon SQS) queue, you need to create an event source mapping. An event source mapping is a resource in Lambda that reads items from a stream or a queue and creates events containing batches of items to send to your Lambda function. Each event that your function processes can contain hundreds or thousands of items.

You can create an event source mapping for your Lambda function using the Lambda console, the AWS CLI, the Lambda API, or an AWS SDK. To create an event source mapping in the Lambda
console, follow the instructions to create a trigger, and select one of the AWS services that support event source mappings as your source. To create an event source mapping using the AWS CLI, Lambda API, or an AWS SDK, and to see a list of the AWS services which event source mappings can be used with, refer to Lambda event source mappings.

Depending on how your function is invoked, scaling behavior and the types of errors that occur can vary. When you invoke a function synchronously, you receive errors in the response and can retry. When you invoke asynchronously, use an event source mapping, or configure another service to invoke your function, the retry requirements and the way that your function scales to handle large numbers of events will vary. For more information, see Error handling and automatic retries in AWS Lambda.

Topics

- Testing Lambda functions in the console
- Synchronous invocation
- Asynchronous invocation
- Lambda event source mappings
- Lambda event filtering
- Lambda function states
- Error handling and automatic retries in AWS Lambda
- Invoking functions defined as container images
- Lambda recursive loop detection
- Lambda function URLs
Testing Lambda functions in the console

You can test your Lambda function in the console by invoking your function with a test event. A test event is a JSON input to your function. If your function doesn't require input, the event can be an empty document ({}).

When you run a test in the console, Lambda synchronously invokes your function with the test event. The function runtime converts the event JSON into an object and passes it to your code's handler method for processing.

Create a test event

Before you can test in the console, you need to create a private or shareable test event.

Invoking functions with test events

To test a function

1. Open the Functions page of the Lambda console.
2. Choose the name of the function that you want to test.
3. Choose the Test tab.
4. Under Test event, choose Create new event or Edit saved event and then choose the saved event that you want to use.
5. Optionally - choose a Template for the event JSON.
6. Choose Test.
7. To review the test results, under Execution result, expand Details.

To invoke your function without saving your test event, choose Test before saving. This creates an unsaved test event that Lambda preserves only for the duration of the session.

You can also access your saved and unsaved test events on the Code tab. From there, choose Test, and then choose your test event.
Creating private test events

Private test events are available only to the event creator, and they require no additional permissions to use. You can create and save up to 10 private test events per function.

To create a private test event

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of the function that you want to test.
3. Choose the Test tab.
4. Under Test event, do the following:
   a. Choose a Template.
   b. Enter a Name for the test.
   c. In the text entry box, enter the JSON test event.
   d. Under Event sharing settings, choose Private.
5. Choose Save changes.

You can also create new test events on the Code tab. From there, choose Test, Configure test event.

Creating shareable test events

Shareable test events are test events that you can share with other users in the same AWS account. You can edit other users' shareable test events and invoke your function with them.

Lambda saves shareable test events as schemas in an Amazon EventBridge (CloudWatch Events) schema registry named lambda-testevent-schemas. As Lambda utilizes this registry to store and call shareable test events you create, we recommend that you do not edit this registry or create a registry using the lambda-testevent-schemas name.

To see, share, and edit shareable test events, you must have permissions for all of the following EventBridge (CloudWatch Events) schema registry API operations:

- schemas.CreateRegistry
- schemas.CreateSchema
- schemas.DeleteSchema
- `schemas.DeleteSchemaVersion`
- `schemas.DescribeRegistry`
- `schemas.DescribeSchema`
- `schemas.GetDiscoveredSchema`
- `schemas.ListSchemaVersions`
- `schemas.UpdateSchema`

Note that saving edits made to a shareable test event overwrites that event.

If you cannot create, edit, or see shareable test events, check that your account has the required permissions for these operations. If you have the required permissions but still cannot access shareable test events, check for any resource-based policies that might limit access to the EventBridge (CloudWatch Events) registry.

**To create a shareable test event**

1. Open the Functions page of the Lambda console.
2. Choose the name of the function that you want to test.
3. Choose the Test tab.
4. Under Test event, do the following:
   a. Choose a Template.
   b. Enter a Name for the test.
   c. In the text entry box, enter the JSON test event.
   d. Under Event sharing settings, choose Shareable.
5. Choose Save changes.

⚠️ **Use shareable test events with AWS Serverless Application Model.**

You can use AWS SAM to invoke shareable test events. See `sam remote test-event` in the AWS Serverless Application Model Developer Guide.
Deleting shareable test event schemas

When you delete shareable test events, Lambda removes them from the lambda-testeventschemas registry. If you remove the last shareable test event from the registry, Lambda deletes the registry.

If you delete the function, Lambda does not delete any associated shareable test event schemas. You must clean up these resources manually from the EventBridge (CloudWatch Events) console.
Synchronous invocation

When you invoke a function synchronously, Lambda runs the function and waits for a response. When the function completes, Lambda returns the response from the function's code with additional data, such as the version of the function that was invoked. To invoke a function synchronously with the AWS CLI, use the `invoke` command.

```bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --payload '{ "key": "value" }' response.json
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/index.html) in the [AWS Command Line Interface User Guide for Version 2](https://docs.aws.amazon.com/cli/latest/index.html).

You should see the following output:

```json
{
   "ExecutedVersion": "$LATEST",
   "StatusCode": 200
}
```

The following diagram shows clients invoking a Lambda function synchronously. Lambda sends the events directly to the function and sends the function's response back to the invoker.

The payload is a string that contains an event in JSON format. The name of the file where the AWS CLI writes the response from the function is `response.json`. If the function returns an
object or error, the response is the object or error in JSON format. If the function exits without error, the response is null.

Note
Lambda does not wait for external extensions to complete before sending the response. External extensions run as independent processes in the execution environment and continue to run after the function invocation is complete. For more information, see Lambda extensions.

The output from the command, which is displayed in the terminal, includes information from headers in the response from Lambda. This includes the version that processed the event (useful when you use aliases), and the status code returned by Lambda. If Lambda was able to run the function, the status code is 200, even if the function returned an error.

Note
For functions with a long timeout, your client might be disconnected during synchronous invocation while it waits for a response. Configure your HTTP client, SDK, firewall, proxy, or operating system to allow for long connections with timeout or keep-alive settings.

If Lambda isn't able to run the function, the error is displayed in the output.

You should see the following output:

An error occurred (InvalidRequestContentException) when calling the Invoke operation:
Could not parse request body into json: Unrecognized token 'value': was expecting ('true', 'false' or 'null')
at [Source: (byte[])"value"; line: 1, column: 11]

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:
You can use the AWS CLI to retrieve logs for an invocation using the --log-type command option. The response contains a LogResult field that contains up to 4 KB of base64-encoded logs from the invocation.

**Example retrieve a log ID**

The following example shows how to retrieve a log ID from the LogResult field for a function named my-function.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{
   "StatusCode": 200,
   "LogResult":
   "U1RBU1QgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTEzZGt0GnkYS0yOTc0YzV1NGZiMjEgVvmVyc21vb...",
   "ExecutedVersion": "$LATEST"
}
```

**Example decode the logs**

In the same command prompt, use the base64 utility to decode the logs. The following example shows how to retrieve base64-encoded logs for my-function.

```
aws lambda invoke --function-name my-function out --log-type Tail 
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --decode
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/reference/index.html) in the [AWS Command Line Interface User Guide for Version 2](https://docs.aws.amazon.com/cli/latest/index.html).

You should see the following output:
The base64 utility is available on Linux, macOS, and Ubuntu on Windows. macOS users may need to use base64 -D.

For more information about the Invoke API, including a full list of parameters, headers, and errors, see Invoke.

When you invoke a function directly, you can check the response for errors and retry. The AWS CLI and AWS SDK also automatically retry on client timeouts, throttling, and service errors. For more information, see Error handling and automatic retries in AWS Lambda.
Asynchronous invocation

Several AWS services, such as Amazon Simple Storage Service (Amazon S3) and Amazon Simple Notification Service (Amazon SNS), invoke functions asynchronously to process events. When you invoke a function asynchronously, you don't wait for a response from the function code. You hand off the event to Lambda and Lambda handles the rest. You can configure how Lambda handles errors, and can send invocation records to a downstream resource such as Amazon Simple Queue Service (Amazon SQS) or Amazon EventBridge (EventBridge) to chain together components of your application.

Sections

- How Lambda handles asynchronous invocations
- Configuring error handling for asynchronous invocation
- Configuring destinations for asynchronous invocation
- Asynchronous invocation configuration API
- Dead-letter queues

How Lambda handles asynchronous invocations

The following diagram shows clients invoking a Lambda function asynchronously. Lambda queues the events before sending them to the function.
For asynchronous invocation, Lambda places the event in a queue and returns a success response without additional information. A separate process reads events from the queue and sends them to your function. To invoke a function asynchronously, set the invocation type parameter to Event.

```bash
aws lambda invoke \
   --function-name my-function  \
   --invocation-type Event \
   --cli-binary-format raw-in-base64-out \
   --payload '{ "key": "value" }' response.json
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-aws-configure-set.html) in the *AWS Command Line Interface User Guide for Version 2*.

```json
{
   "StatusCode": 202
}
```

The output file (response.json) doesn't contain any information, but is still created when you run this command. If Lambda isn't able to add the event to the queue, the error message appears in the command output.

Lambda manages the function's asynchronous event queue and attempts to retry on errors. If the function returns an error, Lambda attempts to run it two more times, with a one-minute wait between the first two attempts, and two minutes between the second and third attempts. Function errors include errors returned by the function's code and errors returned by the function's runtime, such as timeouts.

If the function doesn't have enough concurrency available to process all events, additional requests are throttled. For throttling errors (429) and system errors (500-series), Lambda returns the event to the queue and attempts to run the function again for up to 6 hours. The retry interval increases exponentially from 1 second after the first attempt to a maximum of 5 minutes. If the queue contains many entries, Lambda increases the retry interval and reduces the rate at which it reads events from the queue.

Even if your function doesn't return an error, it's possible for it to receive the same event from Lambda multiple times because the queue itself is eventually consistent. If the function can't keep up with incoming events, events might also be deleted from the queue without being sent to the
function. Ensure that your function code gracefully handles duplicate events, and that you have enough concurrency available to handle all invocations.

When the queue is very long, new events might age out before Lambda has a chance to send them to your function. When an event expires or fails all processing attempts, Lambda discards it. You can configure error handling for a function to reduce the number of retries that Lambda performs, or to discard unprocessed events more quickly.

You can also configure Lambda to send an invocation record to another service. Lambda supports the following destinations for asynchronous invocation. Note that SQS FIFO queues and SNS FIFO topics are not supported.

- **Amazon SQS** – A standard SQS queue.
- **Amazon SNS** – A standard SNS topic.
- **AWS Lambda** – A Lambda function.
- **Amazon EventBridge** – An EventBridge event bus.

The invocation record contains details about the request and response in JSON format. You can configure separate destinations for events that are processed successfully, and events that fail all processing attempts. Alternatively, you can configure a standard Amazon SQS queue or standard Amazon SNS topic as a dead-letter queue for discarded events. For dead-letter queues, Lambda only sends the content of the event, without details about the response.

If Lambda can't send a record to a destination you have configured, it sends a DestinationDeliveryFailures metric to Amazon CloudWatch. This can happen if your configuration includes an unsupported destination type, such as an Amazon SQS FIFO queue or an Amazon SNS FIFO topic. Delivery errors can also occur due to permissions errors and size limits. For more information on Lambda invocation metrics, see Invocation metrics.

**Note**

To prevent a function from triggering, you can set the function's reserved concurrency to zero. When you set reserved concurrency to zero for an asynchronously invoked function, Lambda begins sending new events to the configured dead-letter queue or the on-failure event destination, without any retries. To process events that were sent while reserved concurrency was set to zero, you must consume the events from the dead-letter queue or the on-failure event destination.
Configuring error handling for asynchronous invocation

Use the Lambda console to configure error handling settings on a function, a version, or an alias.

To configure error handling

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose Configuration and then choose Asynchronous invocation.
5. Configure the following settings.
   - Maximum age of event – The maximum amount of time Lambda retains an event in the asynchronous event queue, up to 6 hours.
   - Retry attempts – The number of times Lambda retries when the function returns an error, between 0 and 2.
6. Choose Save.

When an invocation event exceeds the maximum age or fails all retry attempts, Lambda discards it. To retain a copy of discarded events, configure a failed-event destination.

Configuring destinations for asynchronous invocation

To retain records of asynchronous invocations, add a destination to your function. You can choose to send either successful or failed invocations to a destination. Each function can have multiple destinations, so you can configure separate destinations for successful and failed events. Each record sent to the destination is a JSON document with details about the invocation. Like error handling settings, you can configure destinations on a function, function version, or alias.

⚠️ Note
You can also retain records of failed invocations for certain event source mapping types. For more information, see the section called “Configuring destinations for event source mapping invocations”
The following table lists supported destinations for asynchronous invocation records. For Lambda to successfully send records to your chosen destination, ensure that your function’s **execution role** also contains the relevant permissions. The table also describes how each destination type receives the JSON invocation record.

<table>
<thead>
<tr>
<th>Destination type</th>
<th>Required permission</th>
<th>Destination-specific JSON format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon SQS queue</td>
<td>sqs:SendMessage</td>
<td>Lambda passes the invocation record as the Message to the destination.</td>
</tr>
<tr>
<td>Amazon SNS topic</td>
<td>sns:Publish</td>
<td>Lambda passes the invocation record as the Message to the destination.</td>
</tr>
<tr>
<td>Lambda function</td>
<td>InvokeFunction</td>
<td>Lambda passes the invocation record as the payload to the function.</td>
</tr>
</tbody>
</table>
| EventBridge              | events:PutEvents    | • Lambda passes the invocation record as the detail in the PutEvents call.  
                         |         | • The value for the source event field is lambda.  
                         |         | • The value for the detail-type event field is either "Lambda Function Invocation Result - Success" or "Lambda Function Invocation Result - Failure".  
<pre><code>                     |         | • The resource event field contains the function and destination Amazon Resource Names (ARNs). |
</code></pre>
<table>
<thead>
<tr>
<th>Destination type</th>
<th>Required permission</th>
<th>Destination-specific JSON format</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>• For other event fields, see <a href="https://aws.amazon.com/eventbridge/">Amazon EventBridge events</a>.</td>
</tr>
</tbody>
</table>

The following example shows an invocation record for an event that failed three processing attempts due to a function error. The invocation record contains details about the event, the response, and the reason that the record was sent.

```
{
    "version": "1.0",
    "timestamp": "2019-11-14T18:16:05.568Z",
    "requestContext": {
        "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
        "condition": "RetriesExhausted",
        "approximateInvokeCount": 3
    },
    "requestPayload": {
        "ORDER_IDS": [
            "9e07af03-ce31-4ff3-xmpl-36dce652cb4f",
            "637de236-e7b2-464e-xmpl-baf57f86bb53",
            "a81ddca6-2c35-45c7-xmpl-c3a03a31ed15"
        ]
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "responsePayload": {
        "errorMessage": "RequestId: e4b46cbf-b738-xmpl-8880-a18cdf61200e Process exited before completing request"
    }
}
```

The following steps describe how to configure a destination for a function using the Lambda console.
Configuring a destination for asynchronous invocation records

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Under Function overview, choose Add destination.
4. For Source, choose Asynchronous invocation.
5. For Condition, choose from the following options:
   - **On failure** – Send a record when the event fails all processing attempts or exceeds the maximum age.
   - **On success** – Send a record when the function successfully processes an asynchronous invocation.
6. For **Destination type**, choose the type of resource that receives the invocation record.
7. For **Destination**, choose a resource.
8. Choose **Save**.

When an invocation matches the condition, Lambda sends a JSON document with details about the invocation to the destination.

**Destination-specific JSON format**

- For Amazon SQS and Amazon SNS (SnsDestination and SqsDestination), the invocation record is passed as the Message to the destination.
- For Lambda (LambdaDestination), the invocation record is passed as the payload to the function.
- For EventBridge (EventBridgeDestination), the invocation record is passed as the detail in the PutEvents call. The value for the source event field is lambda. The value for the detail-type event field is either Lambda Function Invocation Result – Success or Lambda Function Invocation Result – Failure. The resource event field contains the function and destination Amazon Resource Names (ARNs). For other event fields, see Amazon EventBridge events.

The following example shows an invocation record for an event that failed three processing attempts due to a function error.
Example invocation record

```json
{
    "version": "1.0",
    "timestamp": "2019-11-14T18:16:05.568Z",
    "requestContext": {
        "requestId": "e4b46cbf-b738-xmpl-8880-a18c6f61200e",
        "condition": "RetriesExhausted",
        "approximateInvokeCount": 3
    },
    "requestPayload": {
        "ORDER_ID": [
            "9e07af03-ce31-4ff3-xmpl-36dce652cb4f",
            "637de236-e7b2-464e-xmpl-baf57f86bb53",
            "a81ddca6-2c35-45c7-xmpl-c3a03a031ed15"
        ]
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "responsePayload": {
        "errorMessage": "RequestId: e4b46cbf-b738-xmpl-8880-a18c6f61200e Process exited before completing request"
    }
}
```

The invocation record contains details about the event, the response, and the reason that the record was sent.

**Tracing requests to destinations**

You can use AWS X-Ray to see a connected view of each request as it's queued, processed by a Lambda function, and passed to the destination service. When you activate X-Ray tracing for a function or a service that invokes a function, Lambda adds an X-Ray header to the request and passes the header to the destination service. Traces from upstream services are automatically linked to traces from downstream Lambda functions and destination services, creating an end-to-end view of the entire application. For more information about tracing, see [Using AWS Lambda with AWS X-Ray](https://aws.amazon.com/lambda/x-ray/).
Asynchronous invocation configuration API

To manage asynchronous invocation settings with the AWS CLI or AWS SDK, use the following API operations.

- **PutFunctionEventInvokeConfig**
- **GetFunctionEventInvokeConfig**
- **UpdateFunctionEventInvokeConfig**
- **ListFunctionEventInvokeConfigs**
- **DeleteFunctionEventInvokeConfig**

To configure asynchronous invocation with the AWS CLI, use the `put-function-event-invoke-config` command. The following example configures a function with a maximum event age of 1 hour and no retries.

```bash
aws lambda put-function-event-invoke-config --function-name error --maximum-event-age-in-seconds 3600 --maximum-retry-attempts 0
```

You should see the following output:

```json
{
    "LastModified": 1573686021.479,
    "MaximumRetryAttempts": 0,
    "MaximumEventAgeInSeconds": 3600,
    "DestinationConfig": {
        "OnSuccess": {},
        "OnFailure": {}
    }
}
```

The `put-function-event-invoke-config` command overwrites any existing configuration on the function, version, or alias. To configure an option without resetting others, use `update-function-event-invoke-config`. The following example configures Lambda to send a record to a standard SQS queue named `destination` when an event can’t be processed.

```bash
aws lambda update-function-event-invoke-config --function-name error
```

```bash
aws lambda update-function-event-invoke-config --function-name error
```
Dead-letter queues

As an alternative to an on-failure destination, you can configure your function with a dead-letter queue to save discarded events for further processing. A dead-letter queue acts the same as an on-failure destination in that it is used when an event fails all processing attempts or expires without being processed. However, a dead-letter queue is part of a function's version-specific configuration, so it is locked in when you publish a version. On-failure destinations also support additional targets and include details about the function's response in the invocation record.

To reprocess events in a dead-letter queue, you can set it as an event source for your Lambda function. Alternatively, you can manually retrieve the events.

You can choose an Amazon SQS standard queue or Amazon SNS standard topic for your dead-letter queue. FIFO queues and Amazon SNS FIFO topics are not supported. If you don't have a queue or topic, create one. Choose the target type that matches your use case.

- **Amazon SQS queue** – A queue holds failed events until they're retrieved. Choose an Amazon SQS standard queue if you expect a single entity, such as a Lambda function or CloudWatch alarm, to process the failed event. For more information, see [Using Lambda with Amazon SQS](https://docs.aws.amazon.com/lambda/latest/dg/using-with-sqs.html).

Create a queue in the [Amazon SQS console](https://console.aws.amazon.com/sqs).
• **Amazon SNS topic** – A topic relays failed events to one or more destinations. Choose an Amazon SNS standard topic if you expect multiple entities to act on a failed event. For example, you can configure a topic to send events to an email address, a Lambda function, and/or an HTTP endpoint. For more information, see *Using AWS Lambda with Amazon SNS*.

Create a topic in the [Amazon SNS console](https://console.aws.amazon.com/sns/home).

To send events to a queue or topic, your function needs additional permissions. Add a policy with the required permissions to your function's **execution role**.

• **Amazon SQS** – `sqs:SendMessage`
• **Amazon SNS** – `sns:Publish`

If the target queue or topic is encrypted with a customer managed key, the execution role must also be a user in the key's **resource-based policy**.

After creating the target and updating your function's execution role, add the dead-letter queue to your function. You can configure multiple functions to send events to the same target.

**To configure a dead-letter queue**

1. Open the [Functions page](https://console.aws.amazon.com/lambda/home) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Asynchronous invocation**.
4. Under **Asynchronous invocation**, choose **Edit**.
5. Set **DLQ resource** to **Amazon SQS** or **Amazon SNS**.
6. Choose the target queue or topic.
7. Choose **Save**.

To configure a dead-letter queue with the AWS CLI, use the `update-function-configuration` command.

```
```
Lambda sends the event to the dead-letter queue as-is, with additional information in attributes. You can use this information to identify the error that the function returned, or to correlate the event with logs or an AWS X-Ray trace.

**Dead-letter queue message attributes**

- **RequestID** (String) – The ID of the invocation request. Request IDs appear in function logs. You can also use the X-Ray SDK to record the request ID on an attribute in the trace. You can then search for traces by request ID in the X-Ray console. For an example, see the [error processor sample](#).

- **ErrorCode** (Number) – The HTTP status code.

- **ErrorMessage** (String) – The first 1 KB of the error message.

If Lambda can't send a message to the dead-letter queue, it deletes the event and emits the **DeadLetterErrors** metric. This can happen because of lack of permissions, or if the total size of the message exceeds the limit for the target queue or topic. For example, say that an Amazon SNS notification with a body close to 256 KB in size triggers a function that results in an error. In that case, the event data that Amazon SNS adds, combined with the attributes that Lambda adds, can cause the message to exceed the maximum size allowed in the dead-letter queue.
If you're using Amazon SQS as an event source, configure a dead-letter queue on the Amazon SQS queue itself and not on the Lambda function. For more information, see Using Lambda with Amazon SQS.
Lambda event source mappings

Note
If you want to send data to a target other than a Lambda function or enrich the data before sending it, see Amazon EventBridge Pipes.

An event source mapping is a Lambda resource that reads from an event source and invokes a Lambda function. You can use event source mappings to process items from a stream or queue in services that don't invoke Lambda functions directly. This page describes the services that Lambda provides event source mappings and how-to fine tune batching behavior.

Services that Lambda reads events from

- Amazon DynamoDB
- Amazon Kinesis
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Self-managed Apache Kafka
- Amazon Simple Queue Service (Amazon SQS)
- Amazon DocumentDB (with MongoDB compatibility) (Amazon DocumentDB)

An event source mapping uses permissions in the function's execution role to read and manage items in the event source. Permissions, event structure, settings, and polling behavior vary by event source. For more information, see the linked topic for the service that you use as an event source.

To manage an event source with the AWS Command Line Interface (AWS CLI) or an AWS SDK, you can use the following API operations:

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping
Note

When you update, disable, or delete an event source mapping for Amazon MQ, Amazon MSK, self-managed Apache Kafka, or Amazon DocumentDB, it can take up to 15 minutes for your changes to take effect. Before this period has elapsed, your event source mapping may continue to process events and invoke your function using your previous settings. This is true even when the status of the event source mapping displayed in the console indicates that your changes have been applied.

The following example uses the AWS CLI to map a function named `my-function` to a DynamoDB stream that its Amazon Resource Name (ARN) specifies, with a batch size of 500.

```
```

You should see the following output:

```
{
  "UUID": "14e0db71-5d35-4eb5-b481-8945cf9d10c2",
  "BatchSize": 500,
  "MaximumBatchingWindowInSeconds": 5,
  "ParallelizationFactor": 1,
  "LastModified": 1560209851.963,
  "LastProcessingResult": "No records processed",
  "State": "Creating",
  "StateTransitionReason": "User action",
  "DestinationConfig": {},
  "MaximumRecordAgeInSeconds": 604800,
  "BisectBatchOnFunctionError": false,
  "MaximumRetryAttempts": 10000
}
```

Lambda event source mappings process events at least once due to the distributed nature of its pollers. As a result, your Lambda function may receive duplicate events in rare situations. Follow
Best practices for working with AWS Lambda functions and build idempotent functions to avoid issues related to duplicate events.

**Batching behavior**

Event source mappings read items from a target event source. By default, an event source mapping batches records together into a single payload that Lambda sends to your function. To fine-tune batching behavior, you can configure a batching window (MaximumBatchingWindowInSeconds) and a batch size (BatchSize). A batching window is the maximum amount of time to gather records into a single payload. A batch size is the maximum number of records in a single batch. Lambda invokes your function when one of the following three criteria is met:

- **The batching window reaches its maximum value.** Batching window behavior varies depending on the specific event source.
  - For Kinesis, DynamoDB, and Amazon SQS event sources: The default batching window is 0 seconds. This means that Lambda sends batches to your function as quickly as possible. If you configure a MaximumBatchingWindowInSeconds, the next batching window begins as soon as the previous function invocation completes.
  - For Amazon MSK, self-managed Apache Kafka, Amazon MQ, and Amazon DocumentDB event sources: The default batching window is 500 ms. You can configure MaximumBatchingWindowInSeconds to any value from 0 seconds to 300 seconds in increments of seconds. A batching window begins as soon as the first record arrives.

  **Note**
  
  Because you can only change MaximumBatchingWindowInSeconds in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

- **The batch size is met.** The minimum batch size is 1. The default and maximum batch size depend on the event source. For details about these values, see the BatchSize specification for the CreateEventSourceMapping API operation.

- **The payload size reaches 6 MB.** You cannot modify this limit.

The following diagram illustrates these three conditions. Suppose a batching window begins at \( t = 7 \) seconds. In the first scenario, the batching window reaches its 40 second maximum at \( t = 47 \).
seconds after accumulating 5 records. In the second scenario, the batch size reaches 10 before the batching window expires, so the batching window ends early. In the third scenario, the maximum payload size is reached before the batching window expires, so the batching window ends early.
Max Batching Window = 40 Seconds
Max Batch Size = 10
Max Batch Size in Bytes = 6 MB

(1) Batching Window Expires

(2) Batching Size is reached

(3) Batch Size in bytes is reached
The following example shows an event source mapping that reads from a Kinesis stream. If a batch of events fails all processing attempts, the event source mapping sends details about the batch to an SQS queue.

The event batch is the event that Lambda sends to the function. It is a batch of records or messages compiled from the items that the event source mapping reads up until the current batching window expires.

For Kinesis and DynamoDB streams, an event source mapping creates an iterator for each shard in the stream and processes items in each shard in order. You can configure the event source mapping to read only new items that appear in the stream, or to start with older items. Processed items aren't removed from the stream, and other functions or consumers can process them.

Lambda doesn't wait for any configured **Lambda extensions** to complete before sending the next batch for processing. In other words, your extensions may continue to run as Lambda processes the next batch of records. This can cause throttling issues if you breach any of your account's
concurrency settings or limits. To detect whether this is a potential issue, monitor your functions and check whether you're seeing higher concurrency metrics than expected for your event source mapping. Due to short times in between invokes, Lambda may briefly report higher concurrency usage than the number of shards. This can be true even for Lambda functions without extensions.

By default, if your function returns an error, the event source mapping reprocesses the entire batch until the function succeeds, or the items in the batch expire. To ensure in-order processing, the event source mapping pauses processing for the affected shard until the error is resolved. You can configure the event source mapping to discard old events or process multiple batches in parallel. If you process multiple batches in parallel, in-order processing is still guaranteed for each partition key, but the event source mapping simultaneously processes multiple partition keys in the same shard.

For stream sources (DynamoDB and Kinesis), you can configure the maximum number of times that Lambda retries when your function returns an error. Service errors or throttles where the batch does not reach your function do not count toward retry attempts.

You can also configure the event source mapping to send an invocation record to another service when it discards an event batch. Lambda supports the following destinations for event source mappings.

- **Amazon SQS** – An SQS queue.
- **Amazon SNS** – An SNS topic.

The invocation record contains details about the failed event batch in JSON format.

The following example shows an invocation record for a Kinesis stream.

**Example invocation record**

```json
{
    "requestContext": {
        "requestId": "c9b8fa9f-5a7f-xmpl-af9c-0c604cde93a5",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
```
Lambda also supports in-order processing for FIFO (first-in, first-out) queues, scaling up to the number of active message groups. For standard queues, items aren't necessarily processed in order. Lambda scales up to process a standard queue as quickly as possible. When an error occurs, Lambda returns batches to the queue as individual items and might process them in a different grouping than the original batch. Occasionally, the event source mapping might receive the same item from the queue twice, even if no function error occurred. Lambda deletes items from the queue after they're processed successfully. You can configure the source queue to send items to a dead-letter queue or a destination if Lambda can't process them.

For information about services that invoke Lambda functions directly, see Using AWS Lambda with other services.

### Configuring destinations for event source mapping invocations

To retain records of failed event source mapping invocations, add a destination to your function's event source mapping. Configuring destinations for event source mapping invocations is supported for Kinesis, DynamoDB, and Kafka-based event sources only. Each record sent to the destination is a JSON document with details about the invocation. Like error handling settings, you can configure destinations on a function, function version, or alias.

**Note**

For event source mapping invocations, you can retain records for failed invocations only. For other asynchronous invocations, you can retain records for both successful and failed
You can configure any Amazon SNS topic or any Amazon SQS queue as a destination. For these destination types, Lambda sends the record metadata to the destination. For Kafka-based event sources only, you can also choose an Amazon S3 bucket as the destination. If you specify an S3 bucket, Lambda sends the entire invocation record along with the metadata to the destination.

The following table summarizes the types of supported destinations for event source mapping invocations. For Lambda to successfully send records to your chosen destination, ensure that your function's execution role also contains the relevant permissions. The table also describes how each destination type receives the JSON invocation record.

<table>
<thead>
<tr>
<th>Destination type</th>
<th>Supported for the following event sources</th>
<th>Required permissions</th>
<th>Destination-specific JSON format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon SQS queue</td>
<td>• Kinesis</td>
<td>• sqs:SendMessage</td>
<td>Lambda passes the invocation record metadata as the Message to the destination.</td>
</tr>
<tr>
<td></td>
<td>• DynamoDB</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Self-managed Apache Kafka and Managed Apache Kafka</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon SNS topic</td>
<td>• Kinesis</td>
<td>• sns:Publish</td>
<td>Lambda passes the invocation record metadata as the Message to the destination.</td>
</tr>
<tr>
<td></td>
<td>• DynamoDB</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Self-managed Apache Kafka and Managed Apache Kafka</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon S3 bucket</td>
<td>• Self-managed Apache Kafka and Managed Apache Kafka</td>
<td>• s3:PutObject • s3:ListBuckets</td>
<td>Lambda stores the invocation record along with its metadata at the destination.</td>
</tr>
</tbody>
</table>
The following example shows what Lambda sends to an SQS queue or SNS topic for a failed Kinesis event source invocation. Since Lambda sends only the metadata for these destination types, use the `streamArn`, `shardId`, `startSequenceNumber`, and `endSequenceNumber` fields to obtain the full original record.

```
{
    "requestContext": {
        "requestId": "c9b8fa9f-5a7f-xmpl-af9c-0c604cde93a5",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KinesisBatchInfo": {
        "shardId": "shardId-000000000001",
        "startSequenceNumber": "49601189658422359378836298521827638475320189012309704722",
        "endSequenceNumber": "49601189658422359378836298522902373528957594348623495186",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:38:04.835Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:38:05.580Z",
        "batchSize": 500,
        "streamArn": "arn:aws:kinesis:us-east-2:123456789012:stream/mystream"
    }
}
```

For an example for DynamoDB event sources, see Error handling. For an example for Kafka event sources, see on-failure destinations for self-managed Apache Kafka, or on-failure destinations for Amazon MSK.
Lambda event filtering

You can use event filtering to control which records from a stream or queue Lambda sends to your function. For example, you can add a filter so that your function only processes Amazon SQS messages containing certain data parameters. Event filtering works with event source mappings. You can add filters to event source mappings for the following AWS services:

- Amazon DynamoDB
- Amazon Kinesis Data Streams
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Self-managed Apache Kafka
- Amazon Simple Queue Service (Amazon SQS)

Lambda doesn't support event filtering for Amazon DocumentDB.

By default, you can define up to five different filters for a single event source mapping. Your filters are logically ORed together. If a record from your event source satisfies one or more of your filters, Lambda includes the record in the next event it sends to your function. If none of your filters are satisfied, Lambda discards the record.

Note

If you need to define more than five filters for an event source, you can request a quota increase up to 10 filters for each event source. If you attempt to add more filters than your current quota permits, Lambda will return an error when you try and create the event source.

Topics

- Event filtering basics
- Handling records that don't meet filter criteria
- Filter rule syntax
- Attaching filter criteria to an event source mapping (console)
Event filtering basics

A filter criteria (FilterCriteria) object is a structure that consists of a list of filters (Filters). Each filter is a structure that defines an event filtering pattern (Pattern). A pattern is a string representation of a JSON filter rule. The structure of a FilterCriteria object is as follows.

```json
{
  "Filters": [
    {
      "Pattern": "\"Metadata1\": [ rule1 ], \"data\": { \"Data1\": [ rule2 ] }
    }
  ]
}
```

For added clarity, here is the value of the filter's Pattern expanded in plain JSON.

```json
{
  "Metadata1": [ rule1 ],
  "data": {
    "Data1": [ rule2 ]
  }
}
```

Your filter pattern can include metadata properties, data properties, or both. The available metadata parameters and the format of the data parameters vary according to the AWS service which is acting as the event source. For example, suppose your event source mapping receives the following record from an Amazon SQS queue:
Metadata properties are the fields containing information about the event that created the record. In the example Amazon SQS record, the metadata properties include fields such as messageID, eventSourceArn, and awsRegion.

Data properties are the fields of the record containing the data from your stream or queue. In the Amazon SQS event example, the key for the data field is body, and the data properties are the fields City, State, and Temperature.

Different types of event source use different key values for their data fields. To filter on data properties, make sure that you use the correct key in your filter’s pattern. For a list of data filtering keys, and to see examples of filter patterns for each supported AWS service, refer to Using filters with different AWS services.

Event filtering can handle multi-level JSON filtering. For example, consider the following fragment of a record from a DynamoDB stream:

```
"dynamodb": {
  "Keys": {
    "ID": {
      "S": "ABCD"
    }
  }
  "Number": {
    "N": "1234"
  }
},
```
Suppose you want to process only those records where the value of the sort key `Number` is 4567. In this case, your `FilterCriteria` object would look like this:

```json
{
    "Filters": [
        {
            "Pattern": "{ "dynamodb": { "Keys": { "Number": { "N": [ "4567" ] } } } }"
        }
    ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```json
{
    "dynamodb": {
        "Keys": {
            "Number": {
                "N": [ "4567" ]
            }
        }
    }
}
```

**Handling records that don't meet filter criteria**

The way in which records that don’t meet your filter are handled depends on the event source.

- For **Amazon SQS**, if a message doesn’t satisfy your filter criteria, Lambda automatically removes the message from the queue. You don’t have to manually delete these messages in Amazon SQS.

- For **Kinesis** and **DynamoDB**, once your filter criteria processes a record, the streams iterator advances past this record. If the record doesn’t satisfy your filter criteria, you don’t have to manually delete the record from your event source. After the retention period, Kinesis and DynamoDB automatically delete these old records. If you want records to be deleted sooner, see [Changing the Data Retention Period](#).

- For **Amazon MSK**, **self-managed Apache Kafka**, and **Amazon MQ** messages, Lambda drops messages that don't match all fields included in the filter. For self-managed Apache Kafka,
Lambda commits offsets for matched and unmatched messages after successfully invoking the function. For Amazon MQ, Lambda acknowledges matched messages after successfully invoking the function and acknowledges unmatched messages when filtering them.

**Filter rule syntax**

For filter rules, Lambda supports the Amazon EventBridge rules and uses the same syntax as EventBridge. For more information, see Amazon EventBridge event patterns in the Amazon EventBridge User Guide.

The following is a summary of all the comparison operators available for Lambda event filtering.

<table>
<thead>
<tr>
<th>Comparison operator</th>
<th>Example</th>
<th>Rule syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>UserID is null</td>
<td>&quot;UserID&quot;: [ null ]</td>
</tr>
<tr>
<td>Empty</td>
<td>LastName is empty</td>
<td>&quot;LastName&quot;: [&quot;&quot;&quot;]</td>
</tr>
<tr>
<td>Equals</td>
<td>Name is &quot;Alice&quot;</td>
<td>&quot;Name&quot;: [ &quot;Alice&quot; ]</td>
</tr>
<tr>
<td>Equals (ignore case)</td>
<td>Name is &quot;Alice&quot;</td>
<td>&quot;Name&quot;: [ { &quot;equals-ignore-case&quot;: &quot;alice&quot; } ]</td>
</tr>
<tr>
<td>And</td>
<td>Location is &quot;New York&quot; and Day is &quot;Monday&quot;</td>
<td>&quot;Location&quot;: [ &quot;New York&quot; ], &quot;Day&quot;: [&quot;Monday&quot;]</td>
</tr>
<tr>
<td>Or</td>
<td>PaymentType is &quot;Credit&quot; or &quot;Debit&quot;</td>
<td>&quot;PaymentType&quot;: [ &quot;Credit&quot;, &quot;Debit&quot;]</td>
</tr>
<tr>
<td>Or (multiple fields)</td>
<td>Location is &quot;New York&quot;, or Day is &quot;Monday&quot;.</td>
<td>&quot;$or&quot;: [ { &quot;Location&quot;: [ &quot;New York&quot; ] }, { &quot;Day&quot;: [ &quot;Monday&quot; ] } ]</td>
</tr>
<tr>
<td>Not</td>
<td>Weather is anything but &quot;Raining&quot;</td>
<td>&quot;Weather&quot;: [ { &quot;anything-but&quot;: [ &quot;Raining&quot; ] } ]</td>
</tr>
<tr>
<td>Numeric (equals)</td>
<td>Price is 100</td>
<td>&quot;Price&quot;: [ { &quot;numeric&quot;: [ &quot;=&quot;, 100 ] } ]</td>
</tr>
</tbody>
</table>
### Comparison operator

<table>
<thead>
<tr>
<th>Comparison operator</th>
<th>Example</th>
<th>Rule syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric (range)</td>
<td>Price is more than 10, and less than or equal to 20</td>
<td>&quot;Price&quot;: [ { &quot;numeric&quot;: [ &quot;&gt;&quot;, 10, &quot;&lt;=&quot; , 20 ] } ]</td>
</tr>
<tr>
<td>Exists</td>
<td>ProductName exists</td>
<td>&quot;ProductName&quot;: [ { &quot;exists&quot;: true } ]</td>
</tr>
<tr>
<td>Does not exist</td>
<td>ProductName does not exist</td>
<td>&quot;ProductName&quot;: [ { &quot;exists&quot;: false } ]</td>
</tr>
<tr>
<td>Begins with</td>
<td>Region is in the US</td>
<td>&quot;Region&quot;: [ { &quot;prefix&quot;: &quot;us-&quot; } ]</td>
</tr>
<tr>
<td>Ends with</td>
<td>FileName ends with a .png extension.</td>
<td>&quot;FileName&quot;: [ { &quot;suffix&quot;: &quot;.png&quot; } ]</td>
</tr>
</tbody>
</table>

### Note

Like EventBridge, for strings, Lambda uses exact character-by-character matching without case-folding or any other string normalization. For numbers, Lambda also uses string representation. For example, 300, 300.0, and 3.0e2 are not considered equal.

### Attaching filter criteria to an event source mapping (console)

Follow these steps to create a new event source mapping with filter criteria using the Lambda console.

#### To create a new event source mapping with filter criteria (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of a function to create an event source mapping for.
3. Under **Function overview**, choose **Add trigger**.
4. For **Trigger configuration**, choose a trigger type that supports event filtering. For a list of supported services, refer to the list at the beginning of this page.
5. Expand **Additional settings**.
6. Under **Filter criteria**, choose **Add**, and then define and enter your filters. For example, you can enter the following.

   ```json
   { "Metadata" : [ 1, 2 ] }
   ```

   This instructs Lambda to process only the records where field `Metadata` is equal to 1 or 2. You can continue to select **Add** to add more filters up to the maximum allowed number.

7. When you have finished adding your filters, choose **Save**.

When you enter filter criteria using the console, you enter only the filter pattern and don’t need to provide the `Pattern` key or escape quotes. In step 6 of the preceding instructions, `{ "Metadata" : [ 1, 2 ] }` corresponds to the following `FilterCriteria`.

   ```json
   {
       "Filters": [
           {
               "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
           }
       ]
   }
   ```

After creating your event source mapping in the console, you can see the formatted `FilterCriteria` in the trigger details. For more examples of creating event filters using the console, see [Using filters with different AWS services](#).

### Attaching filter criteria to an event source mapping (AWS CLI)

Suppose you want an event source mapping to have the following `FilterCriteria`:

   ```json
   {
       "Filters": [
           {
               "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
           }
       ]
   }
   ```

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.
aws lambda create-event-source-mapping \
   --function-name my-function \
   --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
   --filter-criteria '{"Filters": [{"Pattern": "{ "Metadata" : [ 1, 2 ]}"}]}'

This `create-event-source-mapping` command creates a new Amazon SQS event source mapping for function `my-function` with the specified FilterCriteria.

To add these filter criteria to an existing event source mapping, run the following command.

aws lambda update-event-source-mapping \
   --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
   --filter-criteria '{"Filters": [{"Pattern": "{ "Metadata" : [ 1, 2 ]}"}]}'

Note that to update an event source mapping, you need its UUID. You can get the UUID from a `list-event-source-mappings` call. Lambda also returns the UUID in the `create-event-source-mapping` CLI response.

To remove filter criteria from an event source, you can run the following `update-event-source-mapping` command with an empty FilterCriteria object.

aws lambda update-event-source-mapping \
   --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
   --filter-criteria "{}"

For more examples of creating event filters using the AWS CLI, see Using filters with different AWS services.

Attaching filter criteria to an event source mapping (AWS SAM)

Suppose you want to configure an event source in AWS SAM to use the following filter criteria:

```json
{
   "Filters": [
      {
         "Pattern": "{ "Metadata" : [ 1, 2 ]}"  
      }
   ]
}
```
To add these filter criteria to your event source mapping, insert the following snippet into the YAML template for your event source.

```yaml
FilterCriteria:
  Filters:
    - Pattern: '{"Metadata": [1, 2]}'
```

For more information on creating and configuring an AWS SAM template for an event source mapping, see the `EventSource` section of the AWS SAM Developer Guide. For more examples of creating event filters using AWS SAM templates, see `Using filters with different AWS services`.

**Using filters with different AWS services**

Different types of event source use different key values for their data fields. To filter on data properties, make sure that you use the correct key in your filter's pattern. The following table gives the filtering keys for each supported AWS service.

<table>
<thead>
<tr>
<th>AWS service</th>
<th>Filtering key</th>
</tr>
</thead>
<tbody>
<tr>
<td>DynamoDB</td>
<td>dynamodb</td>
</tr>
<tr>
<td>Kinesis</td>
<td>data</td>
</tr>
<tr>
<td>Amazon MQ</td>
<td>data</td>
</tr>
<tr>
<td>Amazon MSK</td>
<td>value</td>
</tr>
<tr>
<td>Self-managed Apache Kafka</td>
<td>value</td>
</tr>
<tr>
<td>Amazon SQS</td>
<td>body</td>
</tr>
</tbody>
</table>

The following sections give examples of filter patterns for different types of event source. They also provide definitions of supported incoming data formats and filter pattern body formats for each supported service.
Filtering with DynamoDB

Suppose you have a DynamoDB table with the primary key `CustomerName` and attributes `AccountManager` and `PaymentTerms`. The following shows an example record from your DynamoDB table's stream.

```
{
    "eventID": "1",
    "eventVersion": "1.0",
    "dynamodb": {
        "ApproximateCreationDateTime": "1678831218.0",
        "Keys": {
            "CustomerName": {
                "S": "AnyCompany Industries"
            }
        },
        "NewImage": {
            "AccountManager": {
                "S": "Pat Candella"
            },
            "PaymentTerms": {
                "S": "60 days"
            },
            "CustomerName": {
                "S": "AnyCompany Industries"
            }
        },
        "SequenceNumber": "111",
        "SizeBytes": 26,
        "StreamViewType": "NEW_IMAGE"
    }
}
```

To filter based on the key and attribute values in your DynamoDB table, use the `dynamodb` key in the record. Suppose you want your function to process only those records where the primary key `CustomerName` is “AnyCompany Industries.” The `FilterCriteria` object would be as follows.

```
{
    "Filters": [
        {
            "Pattern": "\"dynamodb\" : { \"Keys\" : { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } }"}
    ]
}
```
For added clarity, here is the value of the filter's Pattern expanded in plain JSON.

```json
{
    "dynamodb": {
        "Keys": {
            "CustomerName": {
                "S": "AnyCompany Industries"
            }
        }
    }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

**Console**

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping (console)](Attaching filter criteria to an event source mapping (console)) and enter the following string for the Filter criteria.

```json
{ "dynamodb": { "Keys": { "CustomerName": { "S": "AnyCompany Industries" } } } }
```

**AWS CLI**

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```bash
aws lambda create-event-source-mapping \
  --function-name my-function \
  --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\": { \"Keys\": { \"CustomerName\": { \"S": [ \"AnyCompany Industries\" ] } } } }]}'}
```

To add these filter criteria to an existing event source mapping, run the following command.

```bash
aws lambda update-event-source-mapping \
  --function-name my-function \
  --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\": { \"Keys\": { \"CustomerName\": { \"S": [ \"AnyCompany Industries\" ] } } } }]}'}
```
AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```yaml
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany Industries" ] } } } }'
```

With DynamoDB, you can also use the NewImage and OldImage keys to filter for attribute values. Suppose you want to filter records where the AccountManager attribute in the latest table image is “Pat Candella” or "Shirley Rodriguez." The FilterCriteria object would be as follows.

```json
{
  "Filters": [
    {
      "Pattern": "{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella", "Shirley Rodriguez" ] } } } }"
    }
  ]
}
```

For added clarity, here is the value of the filter's Pattern expanded in plain JSON.

```json
{
  "dynamodb": {
    "NewImage": {
      "AccountManager": {
        "S": [ "Pat Candella", "Shirley Rodriguez" ]
      }
    }
  }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.
Console

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping (console)] and enter the following string for the Filter criteria.

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella", "Shirley Rodriguez" ] } } } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping \
  --function-name my-function \
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
  --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
```

You can also create filters using Boolean AND expressions. These expressions can include both your table's key and attribute parameters. Suppose you want to filter records where the NewImage
value of AccountManager is "Pat Candella" and the OldImage value is "Terry Whitlock". The FilterCriteria object would be as follows.

```json
{
    "Filters": [
        {
        }
    ]
}
```

For added clarity, here is the value of the filter's Pattern expanded in plain JSON.

```json
{
    "dynamodb": {
        "NewImage": {
            "AccountManager": {
                "S": [ "Pat Candella"
            }
        },
        "dynamodb": {
            "OldImage": {
                "AccountManager": {
                    "S": [ "Terry Whitlock"
                }
            }
        }
    }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

**Console**

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping (console)](Attaching filter criteria to an event source mapping (console)) and enter the following string for the Filter criteria.
AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```bash
cmd
aws lambda create-event-source-mapping
  --function-name my-function
  --filter-criteria '{"Filters": [{"Pattern": "{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" : [ "Terry Whitlock" ] } } } } } ]}'}
```

To add these filter criteria to an existing event source mapping, run the following command.

```bash
cmd
aws lambda update-event-source-mapping
  --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE"
  --filter-criteria '{"Filters": [{"Pattern": "{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" : [ "Terry Whitlock" ] } } } } } ]}'}
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```yaml
FilterCriteria:
  Filters:
```
Note

DynamoDB event filtering doesn't support the use of numeric operators (numeric equals and numeric range). Even if items in your table are stored as numbers, these parameters are converted to strings in the JSON record object.

To properly filter events from DynamoDB sources, both the data field and your filter criteria for the data field (dynamodb) must be in valid JSON format. If either field isn't in a valid JSON format, Lambda drops the message or throws an exception. The following table summarizes the specific behavior:

<table>
<thead>
<tr>
<th>Incoming data format</th>
<th>Filter pattern format for data properties</th>
<th>Resulting action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid JSON</td>
<td>Valid JSON</td>
<td>Lambda filters based on your filter criteria.</td>
</tr>
<tr>
<td>Valid JSON</td>
<td>No filter pattern for data properties</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
<tr>
<td>Valid JSON</td>
<td>Non-JSON</td>
<td>Lambda throws an exception at the time of the event source mapping creation or update. The filter pattern for data properties must be in a valid JSON format.</td>
</tr>
<tr>
<td>Non-JSON</td>
<td>Valid JSON</td>
<td>Lambda drops the record.</td>
</tr>
<tr>
<td>Non-JSON</td>
<td>No filter pattern for data properties</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
<tr>
<td>Non-JSON</td>
<td>Non-JSON</td>
<td>Lambda throws an exception at the time of the event source mapping creation or update. The filter pattern for data properties must be in a valid JSON format.</td>
</tr>
</tbody>
</table>
### Incoming data format

<table>
<thead>
<tr>
<th>Filter pattern format for data properties</th>
<th>Resulting action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>update. The filter pattern for data properties must be in a valid JSON format.</td>
</tr>
</tbody>
</table>

## Filtering with Kinesis

Suppose a producer is putting JSON formatted data into your Kinesis data stream. An example record would look like the following, with the JSON data converted to a Base64 encoded string in the `data` field.

```json
{
   "kinesis": {
      "kinesisSchemaVersion": "1.0",
      "partitionKey": "1",
      "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
      "data": 
        "eyJSZWNvcmRvdW1iZXIiOiAiMDAwMSIsICJUaW1lU3RhbXAiOiAieXl5eS1tbS1kZFRoaDptbTpzcyIsICJSZXF1ZXN0Q29kZSI6ICJBQUFBIn0=
        
        "approximateArrivalTimestamp": 1545084650.987

    },
    "eventSource": "aws:kinesis",
    "eventVersion": "1.0",
    "eventID": "shardId-00000000006:49590338271490256608559692538361571095921575989136588898",
    "eventName": "aws:kinesis:record",
    "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
    "awsRegion": "us-east-2",
    "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"
}
```

As long as the data the producer puts into the stream is valid JSON, you can use event filtering to filter records using the `data` key. Suppose a producer is putting records into your Kinesis stream in the following JSON format.

```json
{
    "record": 12345,
    "order": {
        "type": "buy",
```
To filter only those records where the order type is “buy,” the FilterCriteria object would be as follows.

```
{
    "Filters": [
        {
            "Pattern": "{ "data": { "order": { "type": [ "buy" ] } } }"
        }
    ]
}
```

For added clarity, here is the value of the filter’s Pattern expanded in plain JSON.

```
{
    "data": {
        "order": {
            "type": [ "buy" ]
        }
    }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

**Console**

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping (console)](Attaching%20filter%20criteria%20to%20an%20event%20source%20mapping%20(console)) and enter the following string for the Filter criteria.

```
{ "data" : { "order" : { "type" : [ "buy" ] } } }
```

**AWS CLI**

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping
```
To add these filter criteria to an existing event source mapping, run the following command.

```bash
aws lambda update-event-source-mapping \
  --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
  --filter-criteria '{"Filters": ["Pattern": { "data": { "order": { "type": [ "buy" ] } } }]}'}
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```yaml
FilterCriteria:
  Filters:
    - Pattern: '{ "data": { "order": { "type": [ "buy" ] } } }
```

To properly filter events from Kinesis sources, both the data field and your filter criteria for the data field must be in valid JSON format. If either field isn't in a valid JSON format, Lambda drops the message or throws an exception. The following table summarizes the specific behavior:

<table>
<thead>
<tr>
<th>Incoming data format</th>
<th>Filter pattern format for data properties</th>
<th>Resulting action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid JSON</td>
<td>Valid JSON</td>
<td>Lambda filters based on your filter criteria.</td>
</tr>
<tr>
<td>Valid JSON</td>
<td>No filter pattern for data properties</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
<tr>
<td>Valid JSON</td>
<td>Non-JSON</td>
<td>Lambda throws an exception at the time of the event source mapping creation or</td>
</tr>
</tbody>
</table>

Filtering with Kinesis
## Filter pattern format for data properties

<table>
<thead>
<tr>
<th>Incoming data format</th>
<th>Filter pattern format for data properties</th>
<th>Resulting action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-JSON</td>
<td>Valid JSON</td>
<td>Lambda drops the record.</td>
</tr>
<tr>
<td>Non-JSON</td>
<td>No filter pattern for data properties</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
<tr>
<td>Non-JSON</td>
<td>Non-JSON</td>
<td>Lambda throws an exception at the time of the event source mapping creation or update. The filter pattern for data properties must be in a valid JSON format.</td>
</tr>
</tbody>
</table>

### Filtering Kinesis aggregated records

With Kinesis, you can aggregate multiple records into a single Kinesis Data Streams record to increase your data throughput. Lambda can only apply filter criteria to aggregated records when you use Kinesis enhanced fan-out. Filtering aggregated records with standard Kinesis isn't supported. When using enhanced fan-out, you configure a Kinesis dedicated-throughput consumer to act as the trigger for your Lambda function. Lambda then filters the aggregated records and passes only those records that meet your filter criteria.

To learn more about Kinesis record aggregation, refer to the Aggregation section on the Kinesis Producer Library (KPL) Key Concepts page. To Learn more about using Lambda with Kinesis enhanced fan-out, see Increasing real-time stream processing performance with Amazon Kinesis Data Streams enhanced fan-out and AWS Lambda on the AWS compute blog.
Filtering with Amazon MQ

**Note**

When you update an event source mapping for Amazon MQ with new filter criteria, it can take up to 15 minutes for Lambda to apply your changes. Before this period has elapsed, previous filter settings can still be in effect. This is true even when the status of the event source mapping displayed in the console has changed from Enabling to Enabled.

Suppose your Amazon MQ message queue contains messages either in valid JSON format or as plain strings. An example record would look like the following, with the data converted to a Base64 encoded string in the data field.

ActiveMQ

```
{
   "messageType": "jms/text-message",
   "deliveryMode": 1,
   "replyTo": null,
   "type": null,
   "expiration": "60000",
   "priority": 1,
   "correlationId": "myJMSCoID",
   "redelivered": false,
   "destination": {
      "physicalName": "testQueue"
   },
   "data": "QUJDOkFBQUE=",
   "timestamp": 1598827811958,
   "brokerInTime": 1598827811958,
   "brokerOutTime": 1598827811959,
   "properties": {
      "index": "1",
      "doAlarm": "false",
      "myCustomProperty": "value"
   }
}
```
Filtering with Amazon MQ

```json
{
    "basicProperties": {
        "contentType": "text/plain",
        "contentEncoding": null,
        "headers": {
            "header1": {
                "bytes": [
                    118,
                    97,
                    108,
                    117,
                    101,
                    49
                ]
            },
            "header2": {
                "bytes": [
                    118,
                    97,
                    108,
                    117,
                    101,
                    50
                ]
            },
            "numberInHeader": 10
        },
        "deliveryMode": 1,
        "priority": 34,
        "correlationId": null,
        "replyTo": null,
        "expiration": "60000",
        "messageId": null,
        "timestamp": "Jan 1, 1970, 12:33:41 AM",
        "type": null,
        "userId": "AIDACKCEVSQ6C2EXAMPLE",
        "appId": null,
        "clusterId": null,
        "bodySize": 80
    },
    "redelivered": false,
}
```
For both Active MQ and Rabbit MQ brokers, you can use event filtering to filter records using the data key. Suppose your Amazon MQ queue contains messages in the following JSON format.

```
{
    "timeout": 0,
    "IPAddress": "203.0.113.254"
}
```

To filter only those records where the timeout field is greater than 0, the FilterCriteria object would be as follows.

```
{
    "Filters": [
        {
            "Pattern": "{"data": { "timeout": [{ "numeric": [ "">", 0 ] } ] } }"
        }
    ]
}
```

For added clarity, here is the value of the filter's Pattern expanded in plain JSON.

```
{
    "data": {
        "timeout": [{ "numeric": [ "">", 0 ] ] }
    }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

**Console**

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping (console)] and enter the following string for the Filter criteria.

```
{ "data" : { "timeout" : [ { "numeric" : [ ">", 0 ] } ] } }
```
AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```bash
aws lambda create-event-source-mapping \
   --function-name my-function \
   --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
   --filter-criteria '{"Filters": [{"Pattern": "{ "data": { "timeout": [ { "numeric": [ ">", 0 ] } ] } }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```bash
aws lambda update-event-source-mapping \
   --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
   --filter-criteria '{"Filters": [{"Pattern": "{ "data": { "timeout": [ { "numeric": [ ">", 0 ] } ] } }"}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```yaml
FilterCriteria:
   Filters:
   - Pattern: '{ "data": { "timeout": [ { "numeric": [ ">", 0 ] } ] } }'
```

With Amazon MQ, you can also filter records where the message is a plain string. Suppose you want to process only records where the message begins with "Result: ". The FilterCriteria object would look as follows.

```json
{
   "Filters": [
      {
         "Pattern": "{ \"data\": [ - { \"prefix\": \"Result: \" } ] }"
      }
   ]
}
```
For added clarity, here is the value of the filter's Pattern expanded in plain JSON.

```
{
  "data": [
    {
      "prefix": "Result: 
    }
  ]
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

**Console**

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping (console)](https://docs.aws.amazon.com/lambda/latest/dg/configuration_eventsource_mapping.html) and enter the following string for the Filter criteria.

```
{ "data" : [ { "prefix": "Result: " } ] }
```

**AWS CLI**

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping \
  --function-name my-function \
  --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
  --filter-criteria '{"Filters": [{"Pattern": "{ "data" : [ { "prefix": "Result: " } ] }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
  --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
  --filter-criteria '{"Filters": [{"Pattern": "{ "data" : [ { "prefix": "Result: " } ] }"}]}'
```

**AWS SAM**

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.
Amazon MQ messages must be UTF-8 encoded strings, either plain strings or in JSON format. That's because Lambda decodes Amazon MQ byte arrays into UTF-8 before applying filter criteria. If your messages use another encoding, such as UTF-16 or ASCII, or if the message format doesn't match the FilterCriteria format, Lambda processes metadata filters only. The following table summarizes the specific behavior:

<table>
<thead>
<tr>
<th>Incoming message format</th>
<th>Filter pattern format for message properties</th>
<th>Resulting action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain string</td>
<td>Plain string</td>
<td>Lambda filters based on your filter criteria.</td>
</tr>
<tr>
<td>Plain string</td>
<td>No filter pattern for data properties</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
<tr>
<td>Plain string</td>
<td>Valid JSON</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
<tr>
<td>Valid JSON</td>
<td>Plain string</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
<tr>
<td>Valid JSON</td>
<td>No filter pattern for data properties</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
<tr>
<td>Valid JSON</td>
<td>Valid JSON</td>
<td>Lambda filters based on your filter criteria.</td>
</tr>
<tr>
<td>Incoming message format</td>
<td>Filter pattern format for message properties</td>
<td>Resulting action</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Non-UTF-8 encoded string</td>
<td>JSON, plain string, or no pattern</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
</tbody>
</table>

### Filtering with Amazon MSK and self-managed Apache Kafka

**Note**

When you update an event source mapping for Amazon MSK or self-managed Apache Kafka with new filter criteria, it can take up to 15 minutes for Lambda to apply your changes. Before this period has elapsed, previous filter settings can still be in effect. This is true even when the status of the event source mapping displayed in the console has changed from Enabling to Enabled.

Suppose a producer is writing messages to a topic in your Amazon MSK or self-managed Apache Kafka cluster, either in valid JSON format or as plain strings. An example record would look like the following, with the message converted to a Base64 encoded string in the `value` field.

```json
{
    "mytopic-0":[
        {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[]
        }
    ]
}
```

Suppose your Apache Kafka producer is writing messages to your topic in the following JSON format.
You can use the value key to filter records. Suppose you wanted to filter only those records where device_ID begins with the letters AB. The FilterCriteria object would be as follows.

```json
{
    "Filters": [
        {
            "Pattern": "{ "value" : { "device_ID" : [ { "prefix": "AB" } ] } }"
        }
    ]
}
```

For added clarity, here is the value of the filter's Pattern expanded in plain JSON.

```json
{
    "value": {
        "device_ID": [ { "prefix": "AB" } ]
    }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

**Console**

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping (console)] and enter the following string for the Filter criteria.

```json
{ "value" : { "device_ID" : [ { "prefix": "AB" } ] } }
```

**AWS CLI**

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.
aws lambda create-event-source-mapping \
  --function-name my-function \
  --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
  --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" : [ { \"prefix\" : \"AB\" } ] } }"}]}'

To add these filter criteria to an existing event source mapping, run the following command.

aws lambda update-event-source-mapping \
  --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
  --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" : [ { \"prefix\" : \"AB\" } ] } }"}]}'

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

FilterCriteria:
  Filters:
    - Pattern: '{ "value" : { "device_ID" : [ { "prefix" : "AB" } ] } }'

With Amazon MSK and self-managed Apache Kafka, you can also filter records where the message is a plain string. Suppose you want to ignore those messages where the string is "error". The FilterCriteria object would look as follows.

{
  "Filters": [
    {
      "Pattern": "{ \"value\" : [ { "anything-but" : [ \"error\" ] } ] }"
    }
  ]
}

For added clarity, here is the value of the filter's Pattern expanded in plain JSON.

{
  "value": [ 
    {
    "anything-but": [ "error" ]
  }]"}
You can add your filter using the console, AWS CLI or an AWS SAM template.

**Console**

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping (console)](link) and enter the following string for the Filter criteria.

```
{ "value" : [ { "anything-but": [ "error" ] } ] }
```

**AWS CLI**

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping 
   --function-name my-function 
   --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/b-8ac7cc01-5898-482d-be2f-a6b596050ea8 
   --filter-criteria '{"Filters": [{"Pattern": "{ "value" : [ { "anything-but": [ "error" ] } ] }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping 
   --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" 
   --filter-criteria '{"Filters": [{"Pattern": "{ "value" : [ { "anything-but": [ "error" ] } ] }"}]}'
```

**AWS SAM**

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```yaml
FilterCriteria: 
  Filters: 
    - Pattern: '{ "value" : [ { "anything-but": [ "error" ] } ] }'
```
Amazon MSK and self-managed Apache Kafka messages must be UTF-8 encoded strings, either plain strings or in JSON format. That's because Lambda decodes Amazon MSK byte arrays into UTF-8 before applying filter criteria. If your messages use another encoding, such as UTF-16 or ASCII, or if the message format doesn't match the FilterCriteria format, Lambda processes metadata filters only. The following table summarizes the specific behavior:

<table>
<thead>
<tr>
<th>Incoming message format</th>
<th>Filter pattern format for message properties</th>
<th>Resulting action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain string</td>
<td>Plain string</td>
<td>Lambda filters based on your filter criteria.</td>
</tr>
<tr>
<td>Plain string</td>
<td>No filter pattern for data properties</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
<tr>
<td>Plain string</td>
<td>Valid JSON</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
<tr>
<td>Valid JSON</td>
<td>Plain string</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
<tr>
<td>Valid JSON</td>
<td>No filter pattern for data properties</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
<tr>
<td>Valid JSON</td>
<td>Valid JSON</td>
<td>Lambda filters based on your filter criteria.</td>
</tr>
<tr>
<td>Non-UTF-8 encoded string</td>
<td>JSON, plain string, or no pattern</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
</tbody>
</table>

### Filtering with Amazon SQS

Suppose your Amazon SQS queue contains messages in the following JSON format.
An example record for this queue would look as follows.

```json
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgx1aS3SLy0a...",
    "body": "{\n    "RecordNumber": 0000,\n    "TimeStamp": "yyyy-mm-ddThh:mm:ss",\n    "RequestCode": "AAAA"
   \n},
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIENQZJLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
    },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "awsRegion": "us-west-2"
}
```

To filter based on the contents of your Amazon SQS messages, use the `body` key in the Amazon SQS message record. Suppose you want to process only those records where the `RequestCode` in your Amazon SQS message is “BBBB.” The `FilterCriteria` object would be as follows.

```json
{
    "Filters": [
        {
            "Pattern": "{ \"body\": { \"RequestCode\": [ \"BBBB\" ] } }"
        }
    ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```json
{
    "Filters": [
        {
            "Pattern": "{ \"body\": { \"RequestCode\": [ \"BBBB\" ] } }"
        }
    ]
}
```
You can add your filter using the console, AWS CLI or an AWS SAM template.

**Console**

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping (console)](Attaching filter criteria to an event source mapping (console)) and enter the following string for the **Filter criteria**.

```json
{ "body" : { "RequestCode" : [ "BBBB" ] } }
```

**AWS CLI**

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```bash
aws lambda create-event-source-mapping \
  --function-name my-function \
  --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
  --filter-criteria '{"Filters": ["Pattern": "\"body\" : { \"RequestCode\" : [ \"BBBB\" ] }"]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```bash
aws lambda update-event-source-mapping \
  --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
  --filter-criteria '{"Filters": ["Pattern": "\"body\" : { \"RequestCode\" : [ \"BBBB\" ] }"]}'
```

**AWS SAM**

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```yaml
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RequestCode" : [ "BBBB" ] } }'
```
Suppose you want your function to process only those records where \texttt{RecordNumber} is greater than 9999. The \texttt{FilterCriteria} object would be as follows.

```json
{
    "Filters": [ 
        {
            "Pattern": "{ \"body\": { \"RecordNumber\": [ { \"numeric\": [ \"\", 9999 ] } ] } }"
        }
    ]
}
```

For added clarity, here is the value of the filter's \texttt{Pattern} expanded in plain JSON.

```json
{
    "body": {
        "RecordNumber": [
            {
                "numeric": [ ">", 9999 ]
            }
        ]
    }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

**Console**

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping (console)](Attaching%20filter%20criteria%20to%20an%20event%20source%20mapping%20(console)) and enter the following string for the \texttt{Filter criteria}.

```json
{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }
```

**AWS CLI**

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```bash
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
```
To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
  --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
  --filter-criteria '{"Filters": [{"Pattern": "{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }"}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }'
```

For Amazon SQS, the message body can be any string. However, this can be problematic if your FilterCriteria expect body to be in a valid JSON format. The reverse scenario is also true—if the incoming message body is in JSON format but your filter criteria expects body to be a plain string, this can lead to unintended behavior.

To avoid this issue, ensure that the format of body in your FilterCriteria matches the expected format of body in messages that you receive from your queue. Before filtering your messages, Lambda automatically evaluates the format of the incoming message body and of your filter pattern for body. If there is a mismatch, Lambda drops the message. The following table summarizes this evaluation:

<table>
<thead>
<tr>
<th>Incoming message body format</th>
<th>Filter pattern body format</th>
<th>Resulting action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain string</td>
<td>Plain string</td>
<td>Lambda filters based on your filter criteria.</td>
</tr>
<tr>
<td>Incoming message body format</td>
<td>Filter pattern body format</td>
<td>Resulting action</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Plain string</td>
<td>No filter pattern for data properties</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
<tr>
<td>Plain string</td>
<td>Valid JSON</td>
<td>Lambda drops the message.</td>
</tr>
<tr>
<td>Valid JSON</td>
<td>Plain string</td>
<td>Lambda drops the message.</td>
</tr>
<tr>
<td>Valid JSON</td>
<td>No filter pattern for data properties</td>
<td>Lambda filters (on the other metadata properties only) based on your filter criteria.</td>
</tr>
<tr>
<td>Valid JSON</td>
<td>Valid JSON</td>
<td>Lambda filters based on your filter criteria.</td>
</tr>
</tbody>
</table>
Lambda function states

Lambda includes a state field in the function configuration for all functions to indicate when your function is ready to invoke. State provides information about the current status of the function, including whether you can successfully invoke the function. Function states do not change the behavior of function invocations or how your function runs the code. Function states include:

- **Pending** – After Lambda creates the function, it sets the state to pending. While in pending state, Lambda attempts to create or configure resources for the function, such as VPC or EFS resources. Lambda does not invoke a function during pending state. Any invocations or other API actions that operate on the function will fail.

- **Active** – Your function transitions to active state after Lambda completes resource configuration and provisioning. Functions can only be successfully invoked while active.

- **Failed** – Indicates that resource configuration or provisioning encountered an error.

- **Inactive** – A function becomes inactive when it has been idle long enough for Lambda to reclaim the external resources that were configured for it. When you try to invoke a function that is inactive, the invocation fails and Lambda sets the function to pending state until the function resources are recreated. If Lambda fails to recreate the resources, the function is set to the inactive state.

If you are using SDK-based automation workflows or calling Lambda’s service APIs directly, ensure that you check a function's state before invocation to verify that it is active. You can do this with the Lambda API action `GetFunction`, or by configuring a waiter using the AWS SDK for Java 2.0.

```
aws lambda get-function --function-name my-function --query 'Configuration. [State, LastUpdateStatus]'
```

You should see the following output:

```
[  
  "Active",
  "Successful"
]
```

Functions have two other attributes, `StateReason` and `StateReasonCode`. These provide information and context about the function's state when it is not active for troubleshooting issues.
The following operations fail while function creation is pending:

- **Invoke**
- **UpdateFunctionCode**
- **UpdateFunctionConfiguration**
- **PublishVersion**

**Function states while updating**

Lambda provides additional context for functions undergoing updates with the `LastUpdateStatus` attribute, which can have the following statuses:

- **InProgress** – An update is happening on an existing function. While a function update is in progress, invocations go to the function's previous code and configuration.
- **Successful** – The update has completed. Once Lambda finishes the update, this stays set until a further update.
- **Failed** – The function update has failed. Lambda aborts the update and the function's previous code and configuration remain available.

**Example**

The following is the result of `get-function-configuration` on a function undergoing an update.

```json
{
   "FunctionName": "my-function",
   "Runtime": "nodejs20.x",
   "VpcConfig": {
      "SubnetIds": ["subnet-071f712345678e7c8", "subnet-07fd123456788a036", "subnet-0804f77612345cacf"],
      "SecurityGroupIds": ["sg-085912345678492fb"],
      "VpcId": "vpc-08e1234569e011e83"
   }`
FunctionConfiguration has two other attributes, LastUpdateStatusReason and LastUpdateStatusReasonCode, to help troubleshoot issues with updating.

The following operations fail while an asynchronous update is in progress:

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion
- TagResource
Error handling and automatic retries in AWS Lambda

When you invoke a function, two types of error can occur. Invocation errors occur when the invocation request is rejected before your function receives it. Function errors occur when your function's code or runtime returns an error. Depending on the type of error, the type of invocation, and the client or service that invokes the function, the retry behavior and the strategy for managing errors varies.

Issues with the request, caller, or account can cause invocation errors. Invocation errors include an error type and status code in the response that indicate the cause of the error.

**Common invocation errors**

- **Request** – The request event is too large or isn't valid JSON, the function doesn't exist, or a parameter value is the wrong type.
- **Caller** – The user or service doesn't have permission to invoke the function.
- **Account** – The maximum number of function instances are already running, or requests are being made too quickly.

Clients such as the AWS CLI and the AWS SDK retry on client timeouts, throttling errors (429), and other errors that aren't caused by a bad request. For a full list of invocation errors, see [Invoke](#).

Function errors occur when your function code or the runtime that it uses return an error.

**Common function errors**

- **Function** – Your function's code throws an exception or returns an error object.
- **Runtime** – The runtime terminated your function because it ran out of time, detected a syntax error, or failed to marshal the response object into JSON. The function exited with an error code.

Unlike invocation errors, function errors don't cause Lambda to return a 400-series or 500-series status code. If the function returns an error, Lambda indicates this by including a header named `X-Amz-Function-Error`, and a JSON-formatted response with the error message and other details. For examples of function errors in each language, see the following topics.

- [AWS Lambda function errors in Node.js](#)
- [AWS Lambda function errors in Python](#)
- [AWS Lambda function errors in Ruby](#)
When you invoke a function directly, you determine the strategy for handling errors related to your function code. Lambda does not automatically retry these types of errors on your behalf. To retry, you can manually re-invoke your function, send the failed event to a queue for debugging, or ignore the error. Your function's code might have run completely, partially, or not at all. If you retry, ensure that your function's code can handle the same event multiple times without causing duplicate transactions or other unwanted side effects.

When you invoke a function indirectly, you need to be aware of the retry behavior of the invoker and any service that the request encounters along the way. This includes the following scenarios.

- **Asynchronous invocation** – Lambda retries function errors twice. If the function doesn't have enough capacity to handle all incoming requests, events might wait in the queue for hours or days to be sent to the function. You can configure a dead-letter queue on the function to capture events that weren't successfully processed. For more information, see [Asynchronous invocation](#).

- **Event source mappings** – Event source mappings that read from streams retry the entire batch of items. Repeated errors block processing of the affected shard until the error is resolved or the items expire. To detect stalled shards, you can monitor the [Iterator Age](#) metric.

  For event source mappings that read from a queue, you determine the length of time between retries and destination for failed events by configuring the visibility timeout and redrive policy on the source queue. For more information, see [Lambda event source mappings](#) and the service-specific topics under [Using AWS Lambda with other services](#).

- **AWS services** – AWS services can invoke your function **synchronously** or asynchronously. For synchronous invocation, the service decides whether to retry. For example, Amazon S3 batch operations retries the operation if the Lambda function returns a TemporaryFailure response code. Services that proxy requests from an upstream user or client may have a retry strategy or may relay the error response back to the requestor. For example, API Gateway always relays the error response back to the requestor.

  For asynchronous invocation, the behavior is the same as when you invoke the function synchronously. For more information, see the service-specific topics under [Using AWS Lambda with other services](#) and the invoking service's documentation.
• **Other accounts and clients** – When you grant access to other accounts, you can use resource-based policies to restrict the services or resources they can configure to invoke your function. To protect your function from being overloaded, consider putting an API layer in front of your function with Amazon API Gateway.

To help you deal with errors in Lambda applications, Lambda integrates with services like Amazon CloudWatch and AWS X-Ray. You can use a combination of logs, metrics, alarms, and tracing to quickly detect and identify issues in your function code, API, or other resources that support your application. For more information, see Monitoring and troubleshooting Lambda functions.

For a sample application that uses a CloudWatch Logs subscription, X-Ray tracing, and a Lambda function to detect and process errors, see Error processor sample application for AWS Lambda.
Invoking functions defined as container images

For a Lambda function defined as a container image, function behavior during invocation is very similar to a function defined as a .zip file archive. The following sections highlight the similarities and differences.

Topics

- Function lifecycle
- Invoking the function
- Image security

Function lifecycle

After you upload a new or updated container image, Lambda optimizes the image before the function can process invocations. The optimization process can take a few seconds. The function remains in the Pending state until the process completes. The function then transitions to the Active state. While the state is Pending, you can invoke the function, but other operations on the function fail. Invocations that occur while an image update is in progress run the code from the previous image.

If a function is not invoked for multiple weeks, Lambda reclaims its optimized version, and the function transitions to the Inactive state. To reactivate the function, you must invoke it. Lambda rejects the first invocation and the function enters the Pending state until Lambda re-optimizes the image. The function then returns to the Active state.

Lambda periodically fetches the associated container image from the Amazon Elastic Container Registry (Amazon ECR) repository. If the corresponding container image no longer exists on Amazon ECR or permissions are revoked, the function enters the Failed state, and Lambda returns a failure for any function invocations.

You can use the Lambda API to get information about a function's state. For more information, see Lambda function states.

Invoking the function

When you invoke the function, Lambda deploys the container image to an execution environment. Lambda initializes any extensions and then runs the function's initialization code (the code outside the main handler). Note that function initialization duration is included in billed execution time.
Lambda then runs the function by calling the code entry point specified in the function configuration (the ENTRYPOINT and CMD container image settings).

**Image security**

When Lambda first downloads the container image from its original source (Amazon ECR), the container image is optimized, encrypted, and stored using authenticated convergent encryption methods. All keys that are required to decrypt customer data are protected using AWS KMS customer managed keys. To track and audit Lambda's usage of customer managed keys, you can view the [AWS CloudTrail logs](https://aws.amazon.com/cloudtrail/).
Lambda recursive loop detection

When you configure a Lambda function to output to the same service or resource that invokes the function, it's possible to create an infinite recursive loop. For example, a Lambda function might write a message to an Amazon Simple Queue Service (Amazon SQS) queue, which then invokes the same function. This invocation causes the function to write another message to the queue, which in turn invokes the function again.

Unintentional recursive loops can result in unexpected charges being billed to your AWS account. Loops can also cause Lambda to scale and use all of your account's available concurrency. To help reduce the impact of unintentional loops, Lambda can detect certain types of recursive loops shortly after they occur. When Lambda detects a recursive loop, it stops your function being invoked and notifies you.

If your design intentionally uses recursive patterns, then you can request to turn off Lambda recursive loop detection. To request this change, contact AWS Support.

⚠️ Important

If your design intentionally uses a Lambda function to write data back to the same AWS resource that invokes the function, then use caution and implement suitable guard rails to prevent unexpected charges being billed to your AWS account. To learn more about best practices for using recursive invocation patterns, see Recursive patterns that cause runaway Lambda functions in Serverless Land.

Sections
- Understanding recursive loop detection
- Supported AWS services and SDKs
- Recursive loop notifications
- Responding to recursive loop detection notifications

Understanding recursive loop detection

Recursive loop detection in Lambda works by tracking events. Lambda is an event-driven compute service that runs your function code when certain events occur. For example, when an item is added to an Amazon SQS queue or Amazon Simple Notification Service (Amazon SNS) topic.
Lambda passes events to your function as JSON objects, which contain information about the change in the system state. When an event causes your function to run, this is called an *invocation*.

To detect recursive loops, Lambda uses AWS X-Ray tracing headers. When AWS services that support recursive loop detection send events to Lambda, those events are automatically annotated with metadata. When your Lambda function writes one of these events to another supported AWS service using a supported version of an AWS SDK, it updates this metadata. The updated metadata includes a count of the number of times that the event has invoked the function.

**Note**

You don't need to enable X-Ray active tracing for this feature to work. Recursive loop detection is turned on by default for all AWS customers. There is no charge to use the feature.

A *chain of requests* is a sequence of Lambda invocations caused by the same triggering event. For example, imagine that an Amazon SQS queue invokes your Lambda function. Your Lambda function then sends the processed event back to the same Amazon SQS queue, which invokes your function again. In this example, each invocation of your function falls in the same chain of requests.

If your function is invoked more than 16 times in the same chain of requests, then Lambda automatically stops the next function invocation in that request chain and notifies you. If your function is configured with multiple triggers, then invocations from other triggers aren’t affected.

**Note**

When the maxReceiveCount setting on the source queue's redrive policy is higher than 16, Lambda recursion protection does not prevent Amazon SQS from retrying the message after a recursive loop is detected and terminated. When Lambda detects a recursive loop and drops subsequent invocations, it returns a RecursiveInvocationException to the event source mapping. This increments the maxReceiveCount value. However, Amazon SQS continues to retry the message until maxReceiveCount is exceeded and the message is sent to the dead-letter queue.

If you have an on-failure destination or dead-letter queue configured for your function, then Lambda also sends the event from the stopped invocation to your destination or dead-letter...
queue. When configuring a destination or dead-letter queue for your function, be sure not to use an Amazon SNS topic or Amazon SQS queue that your function also uses as an event trigger or event source mapping. If you send events to the same resource that invokes your function, then you can create another recursive loop.

**Supported AWS services and SDKs**

Lambda can detect only recursive loops that include certain supported AWS services. For recursive loops to be detected, your function must also use one of the supported AWS SDKs.

**Supported AWS services**

Lambda currently detects recursive loops between your functions, Amazon SQS, and Amazon SNS. Lambda also detects loops comprised only of Lambda functions, which may invoke each other synchronously or asynchronously. The following diagrams show some examples of loops that Lambda can detect:

![Diagram of recursive loops involving Lambda functions, Amazon SQS, and Amazon SNS](image)

When another AWS service such as Amazon DynamoDB or Amazon Simple Storage Service (Amazon S3) forms part of the loop, Lambda can't currently detect and stop it.
Because Lambda currently detects only recursive loops involving Amazon SQS and Amazon SNS, it's still possible that loops involving other AWS services can result in unintended usage of your Lambda functions.

To guard against unexpected charges being billed to your AWS account, we recommend that you configure Amazon CloudWatch alarms to alert you to unusual usage patterns. For example, you can configure CloudWatch to notify you about spikes in Lambda function concurrency or invocations. You can also configure a billing alarm to notify you when spending in your account exceeds a threshold that you specify. Or, you can use AWS Cost Anomaly Detection to alert you to unusual billing patterns.

**Supported AWS SDKs**

For Lambda to detect recursive loops, your function must use one of the following SDK versions or higher:

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Minimum required AWS SDK version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node.js</td>
<td>2.1147.0 (SDK version 2)</td>
</tr>
<tr>
<td></td>
<td>3.105.0 (SDK version 3)</td>
</tr>
<tr>
<td>Python</td>
<td>1.24.46 (boto3)</td>
</tr>
<tr>
<td></td>
<td>1.27.46 (botocore)</td>
</tr>
<tr>
<td>Java 8 and Java 11</td>
<td>1.12.200 (SDK version 1)</td>
</tr>
<tr>
<td></td>
<td>2.17.135 (SDK version 2)</td>
</tr>
<tr>
<td>Java 17</td>
<td>2.20.81</td>
</tr>
<tr>
<td>.NET</td>
<td>3.7.293.0</td>
</tr>
<tr>
<td>Ruby</td>
<td>3.134.0</td>
</tr>
</tbody>
</table>

Some Lambda runtimes such as Python and Node.js include a version of the AWS SDK. If the SDK version included in your function's runtime is lower than the minimum required, then you can add a supported version of the SDK to your function's deployment package. You can also add a
supported SDK version to your function using a Lambda layer. For a list of the SDKs included with each Lambda runtime, see Lambda runtimes.

Lambda recursion detection is not supported for the Lambda Go runtime.

**Recursive loop notifications**

When Lambda stops a recursive loop, you receive notifications through the AWS Health Dashboard and through email. You can also use CloudWatch metrics to monitor the number of recursive invocations that Lambda has stopped.

**AWS Health Dashboard notifications**

When Lambda stops a recursive invocation, the AWS Health Dashboard displays a notification on the Your account health page, under Open and recent issues. Note that it can take up to three hours after Lambda stops a recursive invocation before this notification is displayed. For more information about viewing account events in the AWS Health Dashboard, see Getting started with your AWS Health Dashboard – Your account health in the AWS Health User Guide.

**Email alerts**

When Lambda first stops a recursive invocation of your function, it sends you an email alert. Lambda sends a maximum of one email every 24 hours for each function in your AWS account. After Lambda sends an email notification, you won't receive any more emails for that function for another 24 hours, even if Lambda stops further recursive invocations of the function. Note that it can take up to three hours after Lambda stops a recursive invocation before you receive this email alert.

Lambda sends recursive loop email alerts to your AWS account's primary account contact and alternate operations contact. For information about viewing or updating the email addresses in your account, see Updating contact information in the AWS General Reference.

**Amazon CloudWatch metrics**

The CloudWatch metric RecursiveInvocationsDropped records the number of function invocations that Lambda has stopped because your function has been invoked more than 16 times in a single chain of requests. Lambda emits this metric as soon as it stops a recursive invocation. To view this metric, follow the instructions for Viewing metrics on the CloudWatch console and choose the metric RecursiveInvocationsDropped.
Responding to recursive loop detection notifications

When your function is invoked more than 16 times by the same triggering event, Lambda stops the next function invocation for that event to break the recursive loop. To prevent a reoccurrence of a recursive loop that Lambda has broken, do the following:

- Reduce your function's available concurrency to zero, which throttles all future invocations.
- Remove or disable the trigger or event source mapping that's invoking your function.
- Identify and fix code defects that write events back to the AWS resource that's invoking your function. A common source of defects occurs when you use variables to define a function's event source and target. Check that you're not using the same value for both variables.

Additionally, if the event source for your Lambda function is an Amazon SQS queue, then consider configuring a dead-letter queue on the source queue.

Note

Make sure that you configure the dead-letter queue on the source queue, not on the Lambda function. The dead-letter queue that you configure on a function is used for the function's asynchronous invocation queue, not for event source queues.

If the event source is an Amazon SNS topic, then consider adding an on-failure destination for your function.

To reduce your function's available concurrency to zero (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of your function.
3. Choose Throttle.
4. In the Throttle your function dialog box, choose Confirm.

To remove a trigger or event source mapping for your function (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of your function.
3. Choose the **Configuration** tab, then choose **Triggers**.

4. Under **Triggers**, select the trigger or event source mapping that you want to delete, then choose **Delete**.

5. In the **Delete triggers** dialog box, choose **Delete**.

**To disable an event source mapping for your function (AWS CLI)**

1. To find the UUID for the event source mapping that you want to disable, run the AWS Command Line Interface (AWS CLI) `list-event-source-mappings` command.

   ```bash
   aws lambda list-event-source-mappings
   ```

2. To disable the event source mapping, run the following AWS CLI `update-event-source-mapping` command.

   ```bash
   aws lambda update-event-source-mapping --function-name MyFunction \ 
   --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 --no-enabled
   ```
Lambda function URLs

A function URL is a dedicated HTTP(S) endpoint for your Lambda function. You can create and configure a function URL through the Lambda console or the Lambda API. When you create a function URL, Lambda automatically generates a unique URL endpoint for you. Once you create a function URL, its URL endpoint never changes. Function URL endpoints have the following format:

https://<url-id>.lambda-url.<region>.on.aws

**Note**
Function URLs are not supported in the following regions: Asia Pacific (Hyderabad) (ap-south-2), Asia Pacific (Melbourne) (ap-southeast-4), Europe (Spain) (eu-south-2), Europe (Zurich) (eu-central-2), Israel (Tel Aviv) (il-central-1), and Middle East (UAE) (me-central-1).

Function URLs are dual stack-enabled, supporting IPv4 and IPv6. After you configure a function URL for your function, you can invoke your function through its HTTP(S) endpoint via a web browser, curl, Postman, or any HTTP client.

**Note**
You can access your function URL through the public Internet only. While Lambda functions do support AWS PrivateLink, function URLs do not.

Lambda function URLs use [resource-based policies](https://docs.aws.amazon.com/lambda/latest/dg/lambda-security-resource-based-policies.html) for security and access control. Function URLs also support cross-origin resource sharing (CORS) configuration options.

You can apply function URLs to any function alias, or to the $LATEST unpublished function version. You can't add a function URL to any other function version.

**Topics**
- Creating and managing Lambda function URLs
- Security and auth model for Lambda function URLs
- Invoking Lambda function URLs
• Monitoring Lambda function URLs
• Tutorial: Creating a Lambda function with a function URL
Creating and managing Lambda function URLs

A function URL is a dedicated HTTP(S) endpoint for your Lambda function. You can create and configure a function URL through the Lambda console or the Lambda API. When you create a function URL, Lambda automatically generates a unique URL endpoint for you. Once you create a function URL, its URL endpoint never changes. Function URL endpoints have the following format:

https://<url-id>.lambda-url.<region>.on.aws

Note

Function URLs are not supported in the following regions: Asia Pacific (Hyderabad) (ap-south-2), Asia Pacific (Melbourne) (ap-southeast-4), Europe (Spain) (eu-south-2), Europe (Zurich) (eu-central-2), Israel (Tel Aviv) (il-central-1), and Middle East (UAE) (me-central-1).

The following section show how to create and manage a function URL using the Lambda console, AWS CLI, and AWS CloudFormation template

Topics

• Creating a function URL (console)
• Creating a function URL (AWS CLI)
• Adding a function URL to a CloudFormation template
• Cross-origin resource sharing (CORS)
• Throttling function URLs
• Deactivating function URLs
• Deleting function URLs

Creating a function URL (console)

Follow these steps to create a function URL using the console.

To create a function URL for an existing function (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of the function that you want to create the function URL for.

3. Choose the **Configuration** tab, and then choose **Function URL**.

4. Choose **Create function URL**.

5. For **Auth type**, choose **AWS_IAM** or **NONE**. For more information about function URL authentication, see [Security and auth model](#).

6. (Optional) Select **Configure cross-origin resource sharing (CORS)**, and then configure the CORS settings for your function URL. For more information about CORS, see [Cross-origin resource sharing (CORS)](#).

7. Choose **Save**.

This creates a function URL for the $LATEST unpublished version of your function. The function URL appears in the **Function overview** section of the console.

**To create a function URL for an existing alias (console)**

1. Open the **Functions page** of the Lambda console.

2. Choose the name of the function with the alias that you want to create the function URL for.

3. Choose the **Aliases** tab, and then choose the name of the alias that you want to create the function URL for.

4. Choose the **Configuration** tab, and then choose **Function URL**.

5. Choose **Create function URL**.

6. For **Auth type**, choose **AWS_IAM** or **NONE**. For more information about function URL authentication, see [Security and auth model](#).

7. (Optional) Select **Configure cross-origin resource sharing (CORS)**, and then configure the CORS settings for your function URL. For more information about CORS, see [Cross-origin resource sharing (CORS)](#).

8. Choose **Save**.

This creates a function URL for your function alias. The function URL appears in the console's **Function overview** section for your alias.
To create a new function with a function URL (console)

1. Open the Functions page of the Lambda console.
2. Choose Create function.
3. Under Basic information, do the following:
   a. For Function name, enter a name for your function, such as my-function.
   b. For Runtime, choose the language runtime that you prefer, such as Node.js 18.x.
   c. For Architecture, choose either x86_64 or arm64.
   d. Expand Permissions, then choose whether to create a new execution role or use an existing one.
4. Expand Advanced settings, and then select Function URL.
5. For Auth type, choose AWS_IAM or NONE. For more information about function URL authentication, see Security and auth model.
6. (Optional) Select Configure cross-origin resource sharing (CORS). By selecting this option during function creation, your function URL allows requests from all origins by default. You can edit the CORS settings for your function URL after creating the function. For more information about CORS, see Cross-origin resource sharing (CORS).
7. Choose Create function.

This creates a new function with a function URL for the $LATEST unpublished version of the function. The function URL appears in the Function overview section of the console.

Creating a function URL (AWS CLI)

To create a function URL for an existing Lambda function using the AWS Command Line Interface (AWS CLI), run the following command:

```bash
aws lambda create-function-url-config \
  --function-name my-function \
  --qualifier prod \ // optional
  --auth-type AWS_IAM \
  --cors-config {AllowOrigins="https://example.com"} \ // optional
```
This adds a function URL to the **prod** qualifier for the function **my-function**. For more information about these configuration parameters, see [CreateFunctionUrlConfig](#) in the API reference.

### Note

To create a function URL via the AWS CLI, the function must already exist.

## Adding a function URL to a CloudFormation template

To add an `AWS::Lambda::Url` resource to your AWS CloudFormation template, use the following syntax:

### JSON

```
{
   "Type" : "AWS::Lambda::Url",
   "Properties" : {
       "AuthType" : String,
       "Cors" : Cors,
       "Qualifier" : String,
       "TargetFunctionArn" : String
   }
}
```

### YAML

```
Type: AWS::Lambda::Url
Properties:
  AuthType: String
  Cors:
  Cors
  Qualifier: String
  TargetFunctionArn: String
```

### Parameters

- **(AuthType)** – Defines the type of authentication for your function URL. Possible values are either **AWS_IAM** or **NONE**. To restrict access to authenticated users only, set to **AWS_IAM**. To bypass IAM authentication and allow any user to make requests to your function, set to **NONE**.

---

Creating and managing function URLs

304
• (Optional) Cors – Defines the CORS settings for your function URL. To add Cors to your AWS::Lambda::Url resource in CloudFormation, use the following syntax.

Example AWS::Lambda::Url.Cors (JSON)

```json
{
    "AllowCredentials" : Boolean,
    "AllowHeaders" : [ String, ... ],
    "AllowMethods" : [ String, ... ],
    "AllowOrigins" : [ String, ... ],
    "ExposeHeaders" : [ String, ... ],
    "MaxAge" : Integer
}
```

Example AWS::Lambda::Url.Cors (YAML)

```yaml
AllowCredentials: Boolean
AllowHeaders:
    - String
AllowMethods:
    - String
AllowOrigins:
    - String
ExposeHeaders:
    - String
MaxAge: Integer
```

• (Optional) Qualifier – The alias name.

• (Required) TargetFunctionArn – The name or Amazon Resource Name (ARN) of the Lambda function. Valid name formats include the following:

  • **Function name** – my-function
  • **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function
  • **Partial ARN** – 123456789012:function:my-function
## Cross-origin resource sharing (CORS)

To define how different origins can access your function URL, use cross-origin resource sharing (CORS). We recommend configuring CORS if you intend to call your function URL from a different domain. Lambda supports the following CORS headers for function URLs.

<table>
<thead>
<tr>
<th>CORS header</th>
<th>CORS configuration property</th>
<th>Example values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access-Control-Allow-Origin</td>
<td>AllowOrigins</td>
<td>* (allow all origins)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="https://www.example.com">https://www.example.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="http://localhost:60905">http://localhost:60905</a></td>
</tr>
<tr>
<td>Access-Control-Allow-Methods</td>
<td>AllowMethods</td>
<td>GET, POST, DELETE, *</td>
</tr>
<tr>
<td>Access-Control-Allow-Headers</td>
<td>AllowHeaders</td>
<td>Date, Keep-Alive, X-Custom-Header</td>
</tr>
<tr>
<td>Access-Control-Expose-Headers</td>
<td>ExposeHeaders</td>
<td>Date, Keep-Alive, X-Custom-Header</td>
</tr>
<tr>
<td>Access-Control-Allow-Credentials</td>
<td>AllowCredentials</td>
<td>TRUE</td>
</tr>
<tr>
<td>Access-Control-Max-Age</td>
<td>MaxAge</td>
<td>5 (default), 300</td>
</tr>
</tbody>
</table>

When you configure CORS for a function URL using the Lambda console or the AWS CLI, Lambda automatically adds the CORS headers to all responses through the function URL. Alternatively, you can manually add CORS headers to your function response. If there are conflicting headers, the configured CORS headers on the function URL take precedence.
Throttling function URLs

Throttling limits the rate at which your function processes requests. This is useful in many situations, such as preventing your function from overloading downstream resources, or handling a sudden surge in requests.

You can throttle the rate of requests that your Lambda function processes through a function URL by configuring reserved concurrency. Reserved concurrency limits the number of maximum concurrent invocations for your function. Your function's maximum request rate per second (RPS) is equivalent to 10 times the configured reserved concurrency. For example, if you configure your function with a reserved concurrency of 100, then the maximum RPS is 1,000.

Whenever your function concurrency exceeds the reserved concurrency, your function URL returns an HTTP 429 status code. If your function receives a request that exceeds the 10x RPS maximum based on your configured reserved concurrency, you also receive an HTTP 429 error. For more information about reserved concurrency, see Configuring reserved concurrency.

Deactivating function URLs

In an emergency, you might want to reject all traffic to your function URL. To deactivate your function URL, set the reserved concurrency to zero. This throttles all requests to your function URL, resulting in HTTP 429 status responses. To reactivate your function URL, delete the reserved concurrency configuration, or set the configuration to an amount greater than zero.

Deleting function URLs

When you delete a function URL, you can’t recover it. Creating a new function URL will result in a different URL address.

1. Open the Functions page of the Lambda console.
2. Choose the name of the function.
3. Choose the Configuration tab, and then choose Function URL.
4. Choose Delete.
5. Enter the word delete into the field to confirm the deletion.
6. Choose Delete.
Note

If you delete a function URL with auth type NONE, Lambda doesn't automatically delete the associated resource-based policy. If you want to delete this policy, you must manually do so.
Security and auth model for Lambda function URLs

You can control access to your Lambda function URLs using the AuthType parameter combined with resource-based policies attached to your specific function. The configuration of these two components determines who can invoke or perform other administrative actions on your function URL.

The AuthType parameter determines how Lambda authenticates or authorizes requests to your function URL. When you configure your function URL, you must specify one of the following AuthType options:

- **AWS_IAM** – Lambda uses AWS Identity and Access Management (IAM) to authenticate and authorize requests based on the IAM principal's identity policy and the function's resource-based policy. Choose this option if you want only authenticated users and roles to invoke your function via the function URL.

- **NONE** – Lambda doesn't perform any authentication before invoking your function. However, your function's resource-based policy is always in effect and must grant public access before your function URL can receive requests. Choose this option to allow public, unauthenticated access to your function URL.

In addition to AuthType, you can also use resource-based policies to grant permissions to other AWS accounts to invoke your function. For more information, see Using resource-based policies for Lambda.

For additional insights into security, you can use AWS Identity and Access Management Access Analyzer to get a comprehensive analysis of external access to your function URL. IAM Access Analyzer also monitors for new or updated permissions on your Lambda functions to help you identify permissions that grant public and cross-account access. IAM Access Analyzer is free to use for any AWS customer. To get started with IAM Access Analyzer, see Using AWS IAM Access Analyzer.

This page contains examples of resource-based policies for both auth types, and also how to create these policies using the AddPermission API operation or the Lambda console. For information on how to invoke your function URL after you've set up permissions, see Invoking Lambda function URLs.

Topics
Using the AWS_IAM auth type

If you choose the AWS_IAM auth type, users who need to invoke your Lambda function URL must have the `lambda:InvokeFunctionUrl` permission. Depending on who makes the invocation request, you may have to grant this permission using a resource-based policy.

If the principal making the request is in the same AWS account as the function URL, then the principal must either have `lambda:InvokeFunctionUrl` permissions in their identity-based policy, or have permissions granted to them in the function's resource-based policy. In other words, a resource-based policy is optional if the user already has `lambda:InvokeFunctionUrl` permissions in their identity-based policy. Policy evaluation follows the rules outlined in Determining whether a request is allowed or denied within an account.

If the principal making the request is in a different account, then the principal must have both an identity-based policy that gives them `lambda:InvokeFunctionUrl` permissions and permissions granted to them in a resource-based policy on the function that they are trying to invoke. In these cross-account cases, policy evaluation follows the rules outlined in Determining whether a cross-account request is allowed.

For an example cross-account interaction, the following resource-based policy allows the example role in AWS account 444455556666 to invoke the function URL associated with function my-function:

Example function URL cross-account invoke policy

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action": "lambda:InvokeFunctionUrl",
        }
    ]
}
```
"Condition": {
    "StringEquals": {
        "lambda:FunctionUrlAuthType": "AWS_IAM"
    }
}
}

You can create this policy statement through the console by following these steps:

**To grant URL invocation permissions to another account (console)**

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of the function that you want to grant URL invocation permissions for.
3. Choose the Configuration tab, and then choose Permissions.
5. Choose Function URL.
6. For Auth type, choose AWS_IAM.
7. (Optional) For Statement ID, enter a statement ID for your policy statement.
8. For Principal, enter account ID or the Amazon Resource Name (ARN) of the user or role that you want to grant permissions to. For example: 444455556666.
9. Choose Save.

Alternatively, you can create this policy statement using the following `add-permission` AWS Command Line Interface (AWS CLI) command:

```bash
aws lambda add-permission --function-name my-function \
    --statement-id example0-cross-account-statement \
    --action lambda:InvokeFunctionUrl \
    --principal 444455556666 \
    --function-url-auth-type AWS_IAM
```

In the previous example, the `lambda:FunctionUrlAuthType` condition key value is AWS_IAM. This policy only allows access when your function URL's auth type is also AWS_IAM.
Using the NONE auth type

⚠️ Important

When your function URL auth type is NONE and you have a resource-based policy that grants public access, any unauthenticated user with your function URL can invoke your function.

In some cases, you may want your function URL to be public. For example, you might want to serve requests made directly from a web browser. To allow public access to your function URL, choose the NONE auth type.

If you choose the NONE auth type, Lambda doesn't use IAM to authenticate requests to your function URL. However, users must still have `lambda:InvokeFunctionUrl` permissions in order to successfully invoke your function URL. You can grant `lambda:InvokeFunctionUrl` permissions using the following resource-based policy:

Example function URL invoke policy for all unauthenticated principals

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Principal": "*",
         "Action": "lambda:InvokeFunctionUrl",
         "Condition": {
            "StringEquals": {
               "lambda:FunctionUrlAuthType": "NONE"
            }
         }
      }
   ]
}
```

🔍 Note

When you create a function URL with auth type NONE via the console or AWS Serverless Application Model (AWS SAM), Lambda automatically creates the preceding resource-
based policy statement for you. If the policy already exists, or the user or role creating the application doesn't have the appropriate permissions, then Lambda won't create it for you. If you're using the AWS CLI, AWS CloudFormation, or the Lambda API directly, you must add `lambda:InvokeFunctionUrl` permissions yourself. This makes your function public. In addition, if you delete your function URL with auth type `NONE`, Lambda doesn't automatically delete the associated resource-based policy. If you want to delete this policy, you must manually do so.

In this statement, the `lambda:FunctionUrlAuthType` condition key value is `NONE`. This policy statement allows access only when your function URL's auth type is also `NONE`.

If a function's resource-based policy doesn't grant `lambda:invokeFunctionUrl` permissions, then users will get a 403 Forbidden error code when they try to invoke your function URL, even if the function URL uses the `NONE` auth type.

**Governance and access control**

In addition to function URL invocation permissions, you can also control access on actions used to configure function URLs. Lambda supports the following IAM policy actions for function URLs:

- `lambda:InvokeFunctionUrl` – Invoke a Lambda function using the function URL.
- `lambda:CreateFunctionUrlConfig` – Create a function URL and set its `AuthType`.
- `lambda:UpdateFunctionUrlConfig` – Update a function URL configuration and its `AuthType`.
- `lambda:GetFunctionUrlConfig` – View the details of a function URL.
- `lambda:ListFunctionUrlConfigs` – List function URL configurations.
- `lambda:DeleteFunctionUrlConfig` – Delete a function URL.

**Note**

The Lambda console supports adding permissions only for `lambda:InvokeFunctionUrl`. For all other actions, you must add permissions using the Lambda API or AWS CLI.
To allow or deny function URL access to other AWS entities, include these actions in IAM policies. For example, the following policy grants the example role in AWS account 444455556666 permissions to update the function URL for function my-function in account 123456789012.

Example cross-account function URL policy

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action": "lambda:UpdateFunctionUrlConfig",
        }
    ]
}
```

Condition keys

For fine-grained access control over your function URLs, use a condition key. Lambda supports one additional condition key for function URLs: FunctionUrlAuthType. The FunctionUrlAuthType key defines an enum value describing the auth type that your function URL uses. The value can be either AWS_IAM or NONE.

You can use this condition key in policies associated with your function. For example, you might want to restrict who can make configuration changes to your function URLs. To deny all UpdateFunctionUrlConfig requests to any function with URL auth type NONE, you can define the following policy:

Example function URL policy with explicit deny

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": [
```

To grant the example role in AWS account 444455556666 permissions to make CreateFunctionUrlConfig and UpdateFunctionUrlConfig requests on functions with URL auth type AWS_IAM, you can define the following policy:

**Example function URL policy with explicit allow**

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::444455556666:role/example"
      },
      "Action": [
        "lambda:CreateFunctionUrlConfig",
        "lambda:UpdateFunctionUrlConfig"
      ],
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
      "Condition": {
        "StringEquals": {
          "lambda:FunctionUrlAuthType": "AWS_IAM"
        }
      }
    }
  ]
}
```

You can also use this condition key in a service control policy (SCP). Use SCPs to manage permissions across an entire organization in AWS Organizations. For example, to deny users from

Security and auth model
creating or updating function URLs that use anything other than the AWS_IAM auth type, use the following service control policy:

**Example function URL SCP with explicit deny**

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": ["lambda:CreateFunctionUrlConfig", "lambda:UpdateFunctionUrlConfig"],
            "Resource": "arn:aws:lambda:*:123456789012:function:*",
            "Condition": {
                "StringNotEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
}
```
Invoking Lambda function URLs

A function URL is a dedicated HTTP(S) endpoint for your Lambda function. You can create and configure a function URL through the Lambda console or the Lambda API. When you create a function URL, Lambda automatically generates a unique URL endpoint for you. Once you create a function URL, its URL endpoint never changes. Function URL endpoints have the following format:

https://<url-id>.lambda-url.<region>.on.aws

Note
Function URLs are not supported in the following regions: Asia Pacific (Hyderabad) (ap-south-2), Asia Pacific (Melbourne) (ap-southeast-4), Europe (Spain) (eu-south-2), Europe (Zurich) (eu-central-2), Israel (Tel Aviv) (il-central-1), and Middle East (UAE) (me-central-1).

Function URLs are dual stack-enabled, supporting IPv4 and IPv6. After configuring your function URL, you can invoke your function through its HTTP(S) endpoint via a web browser, curl, Postman, or any HTTP client. To invoke a function URL, you must have lambda:InvokeFunctionUrl permissions. For more information, see Security and auth model.

Topics
- Function URL invocation basics
- Request and response payloads

Function URL invocation basics

If your function URL uses the AWS_IAM auth type, you must sign each HTTP request using AWS Signature Version 4 (SigV4). Tools such as awscurl, Postman, and AWS SigV4 Proxy offer built-in ways to sign your requests with SigV4.

If you don't use a tool to sign HTTP requests to your function URL, you must manually sign each request using SigV4. When your function URL receives a request, Lambda also calculates the SigV4 signature. Lambda processes the request only if the signatures match. For instructions on how to manually sign your requests with SigV4, see Signing AWS requests with Signature Version 4 in the Amazon Web Services General Reference Guide.
If your function URL uses the NONE auth type, you don't have to sign your requests using SigV4. You can invoke your function using a web browser, curl, Postman, or any HTTP client.

To test simple GET requests to your function, use a web browser. For example, if your function URL is `https://abcdefg.lambda-url.us-east-1.on.aws`, and it takes in a string parameter `message`, your request URL could look like this:

```
https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld
```

To test other HTTP requests, such as a POST request, you can use a tool such as curl. For example, if you want to include some JSON data in a POST request to your function URL, you could use the following curl command:

```
curl -v 'https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld' \
-H 'content-type: application/json'  \
-d '{ "example": "test" }'
```

**Request and response payloads**

When a client calls your function URL, Lambda maps the request to an event object before passing it to your function. Your function's response is then mapped to an HTTP response that Lambda sends back to the client through the function URL.

The request and response event formats follow the same schema as the [Amazon API Gateway payload format version 2.0](https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-gettarget.html).

**Request payload format**

A request payload has the following structure:

```
{
  "version": "2.0",
  "routeKey": "$default",
  "rawPath": "/my/path",
  "rawQueryString": "parameter1=value1&parameter1=value2&parameter2=value",
  "cookies": [ 
    "cookie1",
    "cookie2"
  ],
  "headers": { 
    "header1": "value1",
    "header2": "value1,value2"
  }
}
```


```
{
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "<urlid>",
    "authentication": null,
    "authorizer": {
      "iam": {
        "accessKey": "AKIA...",
        "accountId": "111122223333",
        "callerId": "AIDA...",
        "cognitoIdentity": null,
        "principalOrgId": null,
        "userArn": "arn:aws:iam::111122223333:user/example-user",
        "userId": "AIDA..."
      }
    },
    "domainName": "<url-id>.lambda-url.us-west-2.on.aws",
    "domainPrefix": "<url-id>",
    "http": {
      "method": "POST",
      "path": "my/path",
      "protocol": "HTTP/1.1",
      "sourceIp": "123.123.123.123",
      "userAgent": "agent"
    },
    "requestId": "id",
    "routeKey": "$default",
    "stage": "$default",
    "time": "12/Mar/2020:19:03:58 +0000",
    "timeEpoch": 1583348638390
  },
  "body": "Hello from client!",
  "pathParameters": null,
  "isBase64Encoded": false,
  "stageVariables": null
}
```
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>The payload format version for this event. Lambda function URLs currently support payload format version 2.0.</td>
<td>2.0</td>
</tr>
<tr>
<td>routeKey</td>
<td>Function URLs don't use this parameter. Lambda sets this to $default as a placeholder.</td>
<td>$default</td>
</tr>
<tr>
<td>rawPath</td>
<td>The request path. For example, if the request URL is <code>https://{url-id}.lambda-url.{region}.on.aws/example/test/demo</code>, then the raw path value is <code>/example/test/demo</code>.</td>
<td>/example/test/demo</td>
</tr>
<tr>
<td>rawQueryString</td>
<td>The raw string containing the request's query string parameters. Supported characters include a-z, A-Z, 0-9, ., _, - , %, &amp;, =, and +.</td>
<td>&quot;?parameter1=value1&amp;parameter2=value2&quot;</td>
</tr>
<tr>
<td>cookies</td>
<td>An array containing all cookies sent as part of the request.</td>
<td>[&quot;Cookie_1=Value_1&quot;, &quot;Cookie_2=Value_2&quot;]</td>
</tr>
<tr>
<td>headers</td>
<td>The list of request headers, presented as key-value pairs.</td>
<td>{&quot;header1&quot;: &quot;value1&quot;, &quot;header2&quot;: &quot;value2&quot;}</td>
</tr>
<tr>
<td>queryStringParameters</td>
<td>The query parameters for the request. For example, if the request URL is <code>https://{url-id}.lambda-url.{region}.on.aws/example/test/demo</code>.</td>
<td>{&quot;name&quot;: &quot;Jane&quot;}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>url-id}.lambda-url.{region}.on.aws/example?name=Jane, then the queryStringParameters value is a JSON object with a key of name and a value of Jane.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>requestContext</td>
<td>An object that contains additional information about the request, such as the requestId, the time of the request, and the identity of the caller if authorized via AWS Identity and Access Management (IAM).</td>
<td></td>
</tr>
<tr>
<td>requestContext.accountId</td>
<td>The AWS account ID of the function owner.</td>
<td>&quot;123456789012&quot;</td>
</tr>
<tr>
<td>requestContext.apiId</td>
<td>The ID of the function URL.</td>
<td>&quot;33anwqw8fj&quot;</td>
</tr>
<tr>
<td>requestContext.authorizer</td>
<td>Function URLs don’t use this parameter. Lambda sets this to null.</td>
<td>null</td>
</tr>
<tr>
<td>requestContext.authorizer.iam.accessKey</td>
<td>An object that contains information about the caller identity, if the function URL uses the AWS_IAM auth type. Otherwise, Lambda sets this to null.</td>
<td></td>
</tr>
<tr>
<td>requestContext.authorizer.iam.accessKey</td>
<td>The access key of the caller identity.</td>
<td>&quot;AKIAIOSFODNN7EXAMPLE&quot;</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td><code>requestContext.authorizer.iam.accountId</code></td>
<td>The AWS account ID of the caller identity.</td>
<td>&quot;111122223333&quot;</td>
</tr>
<tr>
<td><code>requestContext.authorizer.iam.callerId</code></td>
<td>The ID (user ID) of the caller.</td>
<td>&quot;AIDACKCEVSQ6C2EXAMPLE&quot;</td>
</tr>
<tr>
<td><code>requestContext.authorizer.iam.cognitoIdentity</code></td>
<td>Function URLs don't use this parameter. Lambda sets this to null or excludes this from the JSON.</td>
<td>null</td>
</tr>
<tr>
<td><code>requestContext.authorizer.iam.principalOrgId</code></td>
<td>The principal org ID associated with the caller identity.</td>
<td>&quot;AIDACKCEVSQORGEXAMPLE&quot;</td>
</tr>
<tr>
<td><code>requestContext.authorizer.iam.userName</code></td>
<td>The user Amazon Resource Name (ARN) of the caller identity.</td>
<td>&quot;arn:aws:iam::111122223333:user/example-user&quot;</td>
</tr>
<tr>
<td><code>requestContext.authorizer.iam.userId</code></td>
<td>The user ID of the caller identity.</td>
<td>&quot;AIDACOSFODNN7EXAMPLE2&quot;</td>
</tr>
<tr>
<td><code>requestContext.domainName</code></td>
<td>The domain name of the function URL.</td>
<td>&quot;&lt;url-id&gt;.lambda-url.us-west-2.on.aws&quot;</td>
</tr>
<tr>
<td><code>requestContext.domainPrefix</code></td>
<td>The domain prefix of the function URL.</td>
<td>&quot;&lt;url-id&gt;&quot;</td>
</tr>
<tr>
<td><code>requestContext.http</code></td>
<td>An object that contains details about the HTTP request.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>requestContext.http.method</td>
<td>The HTTP method used in this request. Valid values include GET, POST, PUT, HEAD, OPTIONS, PATCH, and DELETE.</td>
<td>GET</td>
</tr>
<tr>
<td>requestContext.http.path</td>
<td>The request path. For example, if the request URL is <code>https://{url-id}.lambda-url.{region}.on.aws/example/test/demo</code>, then the path value is <code>/example/test/demo</code></td>
<td>/example/test/demo</td>
</tr>
<tr>
<td>requestContext.http.protocol</td>
<td>The protocol of the request.</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>requestContext.http.sourceIp</td>
<td>The source IP address of the immediate TCP connection making the request.</td>
<td>123.123.123.123</td>
</tr>
<tr>
<td>requestContext.http.userAgent</td>
<td>The User-Agent request header value.</td>
<td>Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) Gecko/201001 Firefox/42.0</td>
</tr>
<tr>
<td>requestContext.requestId</td>
<td>The ID of the invocation request. You can use this ID to trace invocation logs related to your function.</td>
<td>e1506fd5-9e7b-434f-bd42-4f8fa224b599</td>
</tr>
<tr>
<td>requestContext.routeKey</td>
<td>Function URLs don't use this parameter. Lambda sets this to $default as a placeholder.</td>
<td>$default</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>requestContext.stage</td>
<td>Function URLs don't use this parameter. Lambda sets this to $default as a placeholder.</td>
<td>$default</td>
</tr>
<tr>
<td>requestContext.time</td>
<td>The timestamp of the request.</td>
<td>&quot;07/Sep/2021:22:50:22 +0000&quot;</td>
</tr>
<tr>
<td>requestContext.timeEpoch</td>
<td>The timestamp of the request, in Unix epoch time.</td>
<td>&quot;1631055022677&quot;</td>
</tr>
<tr>
<td>body</td>
<td>The body of the request. If the content type of the request is binary, the body is base64-encoded.</td>
<td>{&quot;key1&quot;: &quot;value1&quot;, &quot;key2&quot;: &quot;value2&quot;}</td>
</tr>
<tr>
<td>pathParameters</td>
<td>Function URLs don't use this parameter. Lambda sets this to null or excludes this from the JSON.</td>
<td>null</td>
</tr>
<tr>
<td>isBase64Encoded</td>
<td>TRUE if the body is a binary payload and base64-encoded. FALSE otherwise.</td>
<td>FALSE</td>
</tr>
<tr>
<td>stageVariables</td>
<td>Function URLs don't use this parameter. Lambda sets this to null or excludes this from the JSON.</td>
<td>null</td>
</tr>
</tbody>
</table>

**Response payload format**

When your function returns a response, Lambda parses the response and converts it into an HTTP response. Function response payloads have the following format:

```
{
```

**Invoking function URLs**
Lambda infers the response format for you. If your function returns valid JSON and doesn't return a statusCode, Lambda assumes the following:

- statusCode is 200.
- content-type is application/json.
- body is the function response.
- isBase64Encoded is false.

The following examples show how the output of your Lambda function maps to the response payload, and how the response payload maps to the final HTTP response. When the client invokes your function URL, they see the HTTP response.

**Example output for a string response**

<table>
<thead>
<tr>
<th>Lambda function output</th>
<th>Interpreted response output</th>
<th>HTTP response (what the client sees)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Hello, world!&quot;</td>
<td>{</td>
<td>HTTP/2 200</td>
</tr>
<tr>
<td></td>
<td>&quot;statusCode&quot;: 200,</td>
<td>date: Wed, 08 Sep 2021 18:02:24 GMT</td>
</tr>
<tr>
<td></td>
<td>&quot;body&quot;: &quot;Hello, world!&quot;,</td>
<td>content-type: application/json</td>
</tr>
<tr>
<td></td>
<td>&quot;headers&quot;: {</td>
<td>content-length: 15</td>
</tr>
<tr>
<td></td>
<td>&quot;content-type&quot;:</td>
<td>&quot;Hello, world!&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;application/json&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;,&quot;isBase64Encoded&quot;:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>Lambda function output</td>
<td>Interpreted response output</td>
<td>HTTP response (what the client sees)</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------</td>
<td>---------------------------------------</td>
</tr>
</tbody>
</table>
| `{  
  "message": "Hello, world!"
} | `{  
  "statusCode": 200,  
  "body": {  
    "message": "Hello, world!"
  },  
  "headers": {  
    "content-type": "application/json"
  },  
  "isBase64Encoded": false
} | HTTP/2 200  
date: Wed, 08 Sep 2021 18:02:24 GMT  
content-type: application/json  
content-length: 34  
{  
  "message": "Hello, world!"
} |

Example output for a JSON response

<table>
<thead>
<tr>
<th>Lambda function output</th>
<th>Interpreted response output</th>
<th>HTTP response (what the client sees)</th>
</tr>
</thead>
</table>
| `{  
  "statusCode": 201,  
  "headers": {  
    "Content-Type": "application/json",  
    "My-Custom-Header": "Custom Value"
  },  
  "body": JSON.stringify({
  "message": "Hello, world!"
})  
} | `{  
  "statusCode": 201,  
  "headers": {  
    "Content-Type": "application/json",  
    "My-Custom-Header": "Custom Value"
  },  
  "body": JSON.stringify({
  "message": "Hello, world!"
})  
} | HTTP/2 201  
date: Wed, 08 Sep 2021 18:02:24 GMT  
content-type: application/json  
content-length: 27  
my-custom-header: Custom Value  
{  
  "message": "Hello, world!"
} |

Example output for a custom response
### Cookies

To return cookies from your function, don't manually add `set-cookie` headers. Instead, include the cookies in your response payload object. Lambda automatically interprets this and adds them as `set-cookie` headers in your HTTP response, as in the following example.

#### Example output for a response returning cookies

<table>
<thead>
<tr>
<th>Lambda function output</th>
<th>HTTP response (what the client sees)</th>
</tr>
</thead>
<tbody>
<tr>
<td>`{ &quot;statusCode&quot;: 201,</td>
<td>HTTP/2 201</td>
</tr>
<tr>
<td>&quot;headers&quot;: {</td>
<td>date: Wed, 08 Sep 2021 18:02:24 GMT</td>
</tr>
<tr>
<td>&quot;Content-Type&quot;: &quot;application/json&quot;,</td>
<td>content-type: application/json</td>
</tr>
<tr>
<td>&quot;My-Custom-Header&quot;: &quot;Custom Value&quot;</td>
<td>content-length: 27</td>
</tr>
<tr>
<td>},</td>
<td>my-custom-header: Custom Value</td>
</tr>
<tr>
<td>&quot;body&quot;: JSON.stringify({</td>
<td>set-cookie: Cookie_1=Value2;</td>
</tr>
<tr>
<td>&quot;message&quot;: &quot;Hello, world!&quot;</td>
<td>Expires=21 Oct 2021 07:48 GMT</td>
</tr>
<tr>
<td>}),</td>
<td>set-cookie: Cookie_2=Value2; Max-Age=78000</td>
</tr>
<tr>
<td>&quot;cookies&quot;: [</td>
<td>{</td>
</tr>
<tr>
<td>&quot;Cookie_1=Value1; Expires=21 Oct 2021 07:48 GMT&quot;,</td>
<td>&quot;message&quot;: &quot;Hello, world!&quot;</td>
</tr>
<tr>
<td>&quot;Cookie_2=Value2; Max-Age=78000&quot;</td>
<td>}</td>
</tr>
<tr>
<td>},</td>
<td></td>
</tr>
<tr>
<td>&quot;isBase64Encoded&quot;: false</td>
<td></td>
</tr>
</tbody>
</table>
Monitoring Lambda function URLs

You can use AWS CloudTrail and Amazon CloudWatch to monitor your function URLs.

Topics

- Monitoring function URLs with CloudTrail
- CloudWatch metrics for function URLs

Monitoring function URLs with CloudTrail

For function URLs, Lambda automatically supports logging the following API operations as events in CloudTrail log files:

- CreateFunctionUrlConfig
- UpdateFunctionUrlConfig
- DeleteFunctionUrlConfig
- GetFunctionUrlConfig
- ListFunctionUrlConfigs

Each log entry contains information about the caller identity, when the request was made, and other details. You can see all events within the last 90 days by viewing your CloudTrail Event history. To retain records past 90 days, you can create a trail. For more information, see Using AWS Lambda with AWS CloudTrail.

By default, CloudTrail doesn't log InvokeFunctionUrl requests, which are considered data events. However, you can turn on data event logging in CloudTrail. For more information, see Logging data events for trails in the AWS CloudTrail User Guide.

CloudWatch metrics for function URLs

Lambda sends aggregated metrics about function URL requests to CloudWatch. With these metrics, you can monitor your function URLs, build dashboards, and configure alarms in the CloudWatch console.

Function URLs support the following invocation metrics. We recommend viewing these metrics with the Sum statistic.
• **UrlRequestCount** – The number of requests made to this function URL.
• **Url4xxCount** – The number of requests that returned a 4XX HTTP status code. 4XX series codes indicate client-side errors, such as bad requests.
• **Url5xxCount** – The number of requests that returned a 5XX HTTP status code. 5XX series codes indicate server-side errors, such as function errors and timeouts.

Function URLs also support the following performance metric. We recommend viewing this metric with the Average or Max statistics.

• **UrlRequestLatency** – The time between when the function URL receives a request and when the function URL returns a response.

Each of these invocation and performance metrics supports the following dimensions:

• **FunctionName** – View aggregate metrics for function URLs assigned to a function's $LATEST unpublished version, or to any of the function's aliases. For example, hello-world-function.
• **Resource** – View metrics for a specific function URL. This is defined by a function name, along with either the function's $LATEST unpublished version or one of the function's aliases. For example, hello-world-function:$LATEST.
• **ExecutedVersion** – View metrics for a specific function URL based on the executed version. You can use this dimension primarily to track the function URL assigned to the $LATEST unpublished version.
Tutorial: Creating a Lambda function with a function URL

In this tutorial, you create a Lambda function defined as a .zip file archive with a public function URL endpoint that returns the product of two numbers. For more information about configuring function URLs, see Creating and managing function URLs.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in Create a Lambda function with the console to create your first Lambda function.

To complete the following steps, you need the AWS Command Line Interface (AWS CLI) version 2. Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system’s built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux. Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

Create an execution role

Create the execution role that gives your Lambda function permission to access AWS resources.
To create an execution role

1. Open the Roles page of the AWS Identity and Access Management (IAM) console.
2. Choose Create role.
3. Create a role with the following properties.
   - Trusted entity – AWS Lambda.
   - Permissions – AWSLambdaBasicExecutionRole.
   - Role name – lambda-url-role.

The AWSLambdaBasicExecutionRole policy has the permissions that the function needs to write logs to Amazon CloudWatch Logs.

Create a Lambda function with a function URL (.zip file archive)

Create a Lambda function with a function URL endpoint using a .zip file archive.

To create the function

1. Copy the following code example into a file named index.js.

Example index.js

```javascript
exports.handler = async (event) => {
  let body = JSON.parse(event.body);
  const product = body.num1 * body.num2;
  const response = {
    statusCode: 200,
    body: "The product of " + body.num1 + " and " + body.num2 + " is " + product,
  }
  return response;
}
```

2. Create a deployment package.

```
zip function.zip index.js
```

3. Create a Lambda function with the create-function command.
**4.** Add a resource-based policy to your function granting permissions to allow public access to your function URL.

```bash
aws lambda add-permission
  --function-name my-url-function
  --action lambda:InvokeFunctionUrl
  --principal "*"
  --function-url-auth-type "NONE"
  --statement-id url
```

**5.** Create a URL endpoint for the function with the `create-function-url-config` command.

```bash
aws lambda create-function-url-config
  --function-name my-url-function
  --auth-type NONE
```

**Test the function URL endpoint**

Invoke your Lambda function by calling your function URL endpoint using an HTTP client such as curl or Postman.

```bash
curl 'https://abcdefg.lambda-url.us-east-1.on.aws/'
-H 'Content-Type: application/json'
-d '{"num1": "10", "num2": "10"}'
```

You should see the following output:

```
The product of 10 and 10 is 100
```
Create a Lambda function with a function URL (CloudFormation)

You can also create a Lambda function with a function URL endpoint using the AWS CloudFormation type `AWS::Lambda::Url`.

Resources:

- **MyUrlFunction**:
  - Type: `AWS::Lambda::Function`
  - Properties:
    - Handler: `index.handler`
    - Runtime: `nodejs18.x`
    - Role: `arn:aws:iam::123456789012:role/lambda-url-role`
    - Code:
      ```
      ZipFile: |
      exports.handler = async (event) => {
        let body = JSON.parse(event.body);
        const product = body.num1 * body.num2;
        const response = {
          statusCode: 200,
          body: "The product of " + body.num1 + " and " + body.num2 + " is " + product,
        };
        return response;
      };
      
      Description: Create a function with a URL.
      ```
  - MyUrlFunctionPermissions:
    - Type: `AWS::Lambda::Permission`
    - Properties:
      - FunctionName: `!Ref MyUrlFunction`
      - Action: `lambda:InvokeFunctionUrl`
      - Principal: "*
      - FunctionUrlAuthType: NONE

- **MyFunctionUrl**:
  - Type: `AWS::Lambda::Url`
  - Properties:
    - TargetFunctionArn: `!Ref MyUrlFunction`
    - AuthType: NONE

Create a Lambda function with a function URL (AWS SAM)

You can also create a Lambda function configured with a function URL using AWS Serverless Application Model (AWS SAM).
ProductFunction:
    Type: AWS::Serverless::Function
Properties:
    CodeUri: function/.
    Handler: index.handler
    Runtime: nodejs18.x
    AutoPublishAlias: live
FunctionUrlConfig:
    AuthType: NONE

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose Delete.
4. Enter the name of the role in the text input field and choose Delete.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose Actions, Delete.
4. Type `delete` in the text input field and choose Delete.
Managing AWS Lambda functions

Learn how to adjust and secure the resources associated with your Lambda function using the Lambda API or console.

**Using Lambda with the AWS CLI**

You can use the AWS Command Line Interface to manage functions and other AWS Lambda resources. The AWS CLI uses the AWS SDK for Python (Boto) to interact with the Lambda API. In this tutorial, you manage and invoke Lambda functions with the AWS CLI.

**Function Scaling**

You can configure two function-level concurrency controls: reserved concurrency and provisioned concurrency. Concurrency is the number of instances of your function that are active and can be configured to ensure critical functions avoid throttling.

**Code signing**

Code signing for Lambda provides trust and integrity controls that let you verify that only unaltered code that approved developers have published is deployed in your Lambda functions.

**Organize with tags**

You can tag Lambda functions to activate attribute-based access control (ABAC) and to organize them by owner, project, or department.

**Using layers**

You can apply previously created layers to reduce deployment package size and promote code sharing and separation of responsibilities so that you can iterate faster on writing business logic.
Using Lambda with the AWS CLI

You can use the AWS Command Line Interface to manage functions and other AWS Lambda resources. The AWS CLI uses the AWS SDK for Python (Boto) to interact with the Lambda API. You can use it to learn about the API, and apply that knowledge in building applications that use Lambda with the AWS SDK.

In this tutorial, you manage and invoke Lambda functions with the AWS CLI. For more information, see What is the AWS CLI? in the AWS Command Line Interface User Guide.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in the section called “Create a Lambda function with the console”.

To complete the following steps, you need the AWS Command Line Interface (AWS CLI) version 2. Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux. Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.
Create the execution role

Create the execution role that gives your function permission to access AWS resources. To create an execution role with the AWS CLI, use the `create-role` command.

In the following example, you specify the trust policy inline. Requirements for escaping quotes in the JSON string vary depending on your shell.

```
aws iam create-role --role-name lambda-ex --assume-role-policy-document '
{"Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Principal": {"Service": "lambda.amazonaws.com"}, "Action": "sts:AssumeRole"}]
}
```

You can also define the trust policy for the role using a JSON file. In the following example, `trust-policy.json` is a file in the current directory. This trust policy allows Lambda to use the role's permissions by giving the service principal `lambda.amazonaws.com` permission to call the AWS Security Token Service (AWS STS) AssumeRole action.

**Example trust-policy.json**

```
{
   "Version": "2012-10-17",
   "Statement": [
   {
      "Effect": "Allow",
      "Principal": {
         "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
   }
   ]
}
```

```
aws iam create-role --role-name lambda-ex --assume-role-policy-document file://trust-policy.json
```

You should see the following output:

```
{
   "Role": {
      "Path": "/",
      "RoleName": "lambda-ex",
   }
}
```
To add permissions to the role, use the `attach-policy-to-role` command. Start by adding the `AWSLambdaBasicExecutionRole` managed policy.

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

The `AWSLambdaBasicExecutionRole` policy has the permissions that the function needs to write logs to CloudWatch Logs.

**Create the function**

The following example logs the values of environment variables and the event object.

**Example index.js**

```javascript
exports.handler = async function(event, context) {
  console.log("ENVIRONMENT VARIABLES
" + JSON.stringify(process.env, null, 2))
  console.log("EVENT
" + JSON.stringify(event, null, 2))
  return context.logStreamName
}
```

**To create the function**

1. Copy the sample code into a file named `index.js`. 

2. Create a deployment package.

```bash
zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command. Replace the highlighted text in the role ARN with your account ID.

```bash
aws lambda create-function --function-name my-function \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs20.x \
--role arn:aws:iam::123456789012:role/lambda-ex
```

You should see the following output:

```json
{
    "FunctionName": "my-function",
    "Runtime": "nodejs20.x",
    "Role": "arn:aws:iam::123456789012:role/lambda-ex",
    "Handler": "index.handler",
    "CodeSha256": "FpFMvUhayLkOoVBpNuNiIVML/tuGv2iJQ7t0yWVTU8c=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "RevisionId": "88ebe1e1-bf6d-8dc3-84de-3017268fa1ff",
    ...
}
```

To get logs for an invocation from the command line, use the `--log-type` option. The response includes a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

```bash
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```json
{
    "StatusCode": 200,
    "LogResult": "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTEzLTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```
You can use the `base64` utility to decode the logs.

```
aws lambda invoke --function-name my-function out --log-type Tail \\n--query 'LogResult' --output text | base64 -d
```

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
  "AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj..., 
  
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Duration: 79.67 ms Billed
  Duration: 80 ms Memory Size: 128 MB Max Memory Used: 73 MB
```

The `base64` utility is available on Linux, macOS, and [Ubuntu on Windows](https://ubuntu.com). For macOS, the command is `base64 -D`.

To get full log events from the command line, you can include the log stream name in the output of your function, as shown in the preceding example. The following example script invokes a function named `my-function` and downloads the last five log events.

**Example get-logs.sh Script**

This example requires that `my-function` returns a log stream ID.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-name
  $(cat out) --limit 5
```

The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to be available. The output includes the response from Lambda and the output from the `get-log-events` command.

```
./get-logs.sh
```
You should see the following output:

```json
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf\tINFO\tENVIRONMENT VARIABLES\r\n\r  "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf\tINFO\tEVENT\r\n\r  "key": "value"
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75 MB",
            "ingestionTime": 1559763003309
        }
    ],
    "nextForwardToken": "f/3478387730485951839386835959492998609206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```
Update the function

After you create a function, you can configure additional capabilities for the function, such as triggers, network access, and file system access. You can also adjust resources associated with the function, such as memory and concurrency. These configurations apply to functions defined as .zip file archives and to functions defined as container images.

Use the `update-function-configuration` command to configure functions. The following example sets the function memory to 256 MB.

**Example update-function-configuration command**

```bash
aws lambda update-function-configuration \
  --function-name my-function \
  --memory-size 256
```

List the Lambda functions in your account

Run the following AWS CLI `list-functions` command to retrieve a list of functions that you have created.

```bash
aws lambda list-functions --max-items 10
```

You should see the following output:

```json
{
  "Functions": [
    {
      "FunctionName": "cli",
      "Runtime": "nodejs20.x",
      "Role": "arn:aws:iam::123456789012:role/lambda-ex",
      "Handler": "index.handler",
      ...
    },
    {
      "FunctionName": "random-error",
      "Runtime": "nodejs20.x",
      ...
    }
  ]
}
```
In response, Lambda returns a list of up to 10 functions. If there are more functions you can retrieve, NextToken provides a marker you can use in the next list-functions request. The following list-functions AWS CLI command is an example that shows the --starting-token parameter.

```
aws lambda list-functions --max-items 10 --starting-token eyJNYXJrZXIiOiBudWxsLCAiYm90b190cnVuY2F0ZV9hbW91bnQiOiAiAxMHi=
```

**Retrieve a Lambda function**

The Lambda CLI get-function command returns Lambda function metadata and a presigned URL that you can use to download the function's deployment package.

```
aws lambda get-function --function-name my-function
```

You should see the following output:

```json
{
  "Configuration": {
    "FunctionName": "my-function",
    "Runtime": "nodejs20.x",
    "Role": "arn:aws:iam::123456789012:role/lambda-ex",
    "CodeSha256": "FpFMvUhayLkOoVBpNuNiIVML/tuGv2iJQ7t0yWVTU8c=",
    "Version": "$LATEST",
    "TracingConfig": {
      "Mode": "PassThrough"
    },
    "RevisionId": "88ebe1e1-bfd5-4dc3-84de-3017268fa1ff",
    ...
  },
  "Code": {
    ...
  }
}
```
"RepositoryType": "S3",
"Location": "https://awslambda-us-east-2-tasks.s3.us-east-2.amazonaws.com/
snapshots/123456789012/my-function-4203078a-b7c9-4f35-..."
}
}

For more information, see `GetFunction`.

**Clean up**

Run the following `delete-function` command to delete the `my-function` function.

```
aws lambda delete-function --function-name my-function
```

Delete the IAM role you created in the IAM console. For information about deleting a role, see `Deleting roles or instance profiles` in the *IAM User Guide*. 
Lambda function scaling

**Concurrency** is the number of in-flight requests that your AWS Lambda function is handling at the same time. For each concurrent request, Lambda provisions a separate instance of your execution environment. As your functions receive more requests, Lambda automatically handles scaling the number of execution environments until you reach your account's concurrency limit. By default, Lambda provides your account with a total concurrency limit of 1,000 concurrent executions across all functions in an AWS Region. To support your specific account needs, you can request a quota increase and configure function-level concurrency controls so that your critical functions don’t experience throttling.

This topic explains concurrency and function scaling in Lambda. By the end of this topic, you'll be able to understand how to calculate concurrency, visualize the two main concurrency control options (reserved and provisioned), estimate appropriate concurrency control settings, and view metrics for further optimization.

**Sections**

- Understanding and visualizing concurrency
- How to calculate concurrency
- Concurrency vs. requests per second
- Reserved concurrency and provisioned concurrency
- Concurrency quotas
- Configuring reserved concurrency
- Configuring provisioned concurrency
- Lambda scaling behavior
- Monitoring concurrency

**Understanding and visualizing concurrency**

Lambda invokes your function in a secure and isolated execution environment. To handle a request, Lambda must first initialize an execution environment (the *Init phase*), before using it to invoke your function (the *Invoke phase*):
Note

Actual Init and Invoke durations can vary depending on many factors, such as the runtime you choose and the Lambda function code. The previous diagram isn't meant to represent the exact proportions of Init and Invoke phase durations.

The previous diagram uses a rectangle to represent a single execution environment. When your function receives its very first request (represented by the yellow circle with label 1), Lambda creates a new execution environment and runs the code outside your main handler during the Init phase. Then, Lambda runs your function's main handler code during the Invoke phase. During this entire process, this execution environment is busy and cannot process other requests.

When Lambda finishes processing the first request, this execution environment can then process additional requests for the same function. For subsequent requests, Lambda doesn't need to re-initialize the environment.

In the previous diagram, Lambda reuses the execution environment to handle the second request (represented by the yellow circle with label 2).
So far, we've focused on just a single instance of your execution environment (that is, a concurrency of 1). In practice, Lambda may need to provision multiple execution environment instances in parallel to handle all incoming requests. When your function receives a new request, one of two things can happen:

- If a pre-initialized execution environment instance is available, Lambda uses it to process the request.
- Otherwise, Lambda creates a new execution environment instance to process the request.

For example, let's explore what happens when your function receives 10 requests:

In the previous diagram, each horizontal plane represents a single execution environment instance (labeled from A through F). Here's how Lambda handles each request:

**Lambda behavior for requests 1 through 10**

<table>
<thead>
<tr>
<th>Request</th>
<th>Lambda behavior</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Provisions new environment A</td>
<td>This is the first request; no execution environment instances are available.</td>
</tr>
<tr>
<td>2</td>
<td>Provisions new environment B</td>
<td>Existing execution environment instance A is busy.</td>
</tr>
<tr>
<td>Request</td>
<td>Lambda behavior</td>
<td>Reasoning</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------</td>
<td>-----------</td>
</tr>
<tr>
<td>3</td>
<td>Provisions new environment C</td>
<td>Existing execution environment instances A and B are both busy.</td>
</tr>
<tr>
<td>4</td>
<td>Provisions new environment D</td>
<td>Existing execution environment instances A, B, and C are all busy.</td>
</tr>
<tr>
<td>5</td>
<td>Provisions new environment E</td>
<td>Existing execution environment instances A, B, C, and D are all busy.</td>
</tr>
<tr>
<td>6</td>
<td>Reuses environment A</td>
<td>Execution environment instance A has finished processing request 1 and is now available.</td>
</tr>
<tr>
<td>7</td>
<td>Reuses environment B</td>
<td>Execution environment instance B has finished processing request 2 and is now available.</td>
</tr>
<tr>
<td>8</td>
<td>Reuses environment C</td>
<td>Execution environment instance C has finished processing request 3 and is now available.</td>
</tr>
<tr>
<td>9</td>
<td>Provisions new environment F</td>
<td>Existing execution environment instances A, B, C, D, and E are all busy.</td>
</tr>
<tr>
<td>10</td>
<td>Reuses environment D</td>
<td>Execution environment instance D has finished processing request 4 and is now available.</td>
</tr>
</tbody>
</table>
As your function receives more concurrent requests, Lambda scales up the number of execution environment instances in response. The following animation tracks the number of concurrent requests over time:

![Animation](image)

By freezing the previous animation at six distinct points in time, we get the following diagram:

![Diagram](image)
In the previous diagram, we can draw a vertical line at any point in time and count the number of environments that intersect this line. This gives us the number of concurrent requests at that point in time. For example, at time $t_1$, there are three active environments serving three concurrent requests. The maximum number of concurrent requests in this simulation occurs at time $t_4$, when there are six active environments serving six concurrent requests.

To summarize, your function's concurrency is the number of concurrent requests that it's handling at the same time. In response to an increase in your function's concurrency, Lambda provisions more execution environment instances to meet request demand.

**How to calculate concurrency**

In general, concurrency of a system is the ability to process more than one task simultaneously. In Lambda, concurrency is the number of in-flight requests that your function is handling at the same time. A quick and practical way of measuring concurrency of a Lambda function is to use the following formula:

$$\text{Concurrency} = (\text{average requests per second}) \times (\text{average request duration in seconds})$$

**Concurrency differs from requests per second.** For example, suppose your function receives 100 requests per second on average. If the average request duration is one second, then it's true that the concurrency is also 100:

$$\text{Concurrency} = (100 \text{ requests/second}) \times (1 \text{ second/request}) = 100$$

However, if the average request duration is 500 ms, then the concurrency is 50:

$$\text{Concurrency} = (100 \text{ requests/second}) \times (0.5 \text{ second/request}) = 50$$

What does a concurrency of 50 mean in practice? If the average request duration is 500 ms, then you can think of an instance of your function as being able to handle two requests per second. Then, it takes 50 instances of your function to handle a load of 100 requests per second. A concurrency of 50 means that Lambda must provision 50 execution environment instances to efficiently handle this workload without any throttling. Here's how to express this in equation form:

$$\text{Concurrency} = (100 \text{ requests/second}) / (2 \text{ requests/second}) = 50$$

If your function receives double the number of requests (200 requests per second), but only requires half the time to process each request (250 ms), then the concurrency is still 50:
Test your understanding of concurrency

Suppose you have a function that takes, on average, 200 ms to run. During peak load, you observe 5,000 requests per second. What is the concurrency of your function during peak load?

Answer

The average function duration is 200 ms, or 0.2 seconds. Using the concurrency formula, you can plug in the numbers to get a concurrency of 1,000:

\[
\text{Concurrency} = (5,000 \text{ requests/second}) \times (0.2 \text{ seconds/request}) = 1,000
\]

Alternatively, an average function duration of 200 ms means that your function can process 5 requests per second. To handle the 5,000 request per second workload, you need 1,000 execution environment instances. Thus, the concurrency is 1,000:

\[
\text{Concurrency} = (5,000 \text{ requests/second}) / (5 \text{ requests/second}) = 1,000
\]

Concurrency vs. requests per second

As mentioned in the previous section, concurrency differs from requests per second. This is an especially important distinction to make when working with functions that have an average request duration of less than 100 ms.

In general, each instance of your execution environment can handle at most 10 requests per second. This limit applies to synchronous on-demand functions, as well as to functions that use provisioned concurrency. If you're unfamiliar with this limit, then you may not know why such functions could experience throttling in certain scenarios.

For example, consider a function with an average request duration of 50 ms. At 200 requests per second, here's the concurrency of this function:

\[
\text{Concurrency} = (200 \text{ requests/second}) \times (0.05 \text{ second/request}) = 10
\]

Based on this result, you might expect that you need only 10 execution environment instances to handle this load. However, each execution environment can handle only 10 executions per second.
This means that with 10 execution environments, your function can handle only 100 requests per second out of the 200 total requests. This function experiences throttling.

The lesson is that you must consider both concurrency and requests per second when configuring concurrency settings for your functions. In this case, you need 20 execution environments for your function, even though it has a concurrency of only 10.

**Test your understanding of concurrency (sub-100 ms functions)**

Suppose that you have a function that takes, on average, 20 ms to run. During peak load, you observe 3,000 requests per second. What is the concurrency of your function during peak load?

**Answer**

The average function duration is 20 ms, or 0.02 seconds. Using the concurrency formula, you can plug in the numbers to get a concurrency of 60:

\[
\text{Concurrency} = (3,000 \text{ requests/second}) \times (0.02 \text{ seconds/request}) = 60
\]

However, each execution environment can handle only 10 requests per second. With 60 execution environments, your function can handle a maximum of 600 requests per second. To fully accommodate the 3,000 requests, your function needs at least 300 execution environment instances.

**Reserved concurrency and provisioned concurrency**

By default, your account has a concurrency limit of 1,000 concurrent executions across all functions in a Region. Your functions share this pool of 1,000 concurrency on an on-demand basis. Your functions experience throttling (that is, they start to drop requests) if you run out of available concurrency.

Some of your functions might be more critical than others. As a result, you might want to configure concurrency settings to ensure that critical functions get the concurrency that they need. There are two types of concurrency controls available: reserved concurrency and provisioned concurrency.

- **Use reserved concurrency** to reserve a portion of your account's concurrency for a function. This is useful if you don't want other functions taking up all the available unreserved concurrency.
- **Use provisioned concurrency** to pre-initialize a number of environment instances for a function. This is useful for reducing cold start latencies.
Reserved concurrency

If you want to guarantee that a certain amount of concurrency is available for your function at any time, use reserved concurrency.

Reserved concurrency is the maximum number of concurrent instances that you want to allocate to your function. When you dedicate reserved concurrency to a function, no other function can use that concurrency. In other words, setting reserved concurrency can impact the concurrency pool that’s available to other functions. Functions that don’t have reserved concurrency share the remaining pool of unreserved concurrency.

Configuring reserved concurrency counts towards your overall account concurrency limit. There is no charge for configuring reserved concurrency for a function.

To better understand reserved concurrency, consider the following diagram:

In this diagram, your account concurrency limit for all the functions in this Region is at the default limit of 1,000. Suppose you have two critical functions, function-blue and function-orange, that routinely expect to get high invocation volumes. You decide to give 400 units of reserved concurrency to function-blue, and 400 units of reserved concurrency to function-orange. In
this example, all other functions in your account must share the remaining 200 units of unreserved concurrency.

The diagram has five points of interest:

• At t1, both function-orange and function-blue begin receiving requests. Each function begins to use up its allocated portion of reserved concurrency units.

• At t2, function-orange and function-blue steadily receive more requests. At the same time, you deploy some other Lambda functions, which begin receiving requests. You don't allocate reserved concurrency to these other functions. They begin using the remaining 200 units of unreserved concurrency.

• At t3, function-orange hits the max concurrency of 400. Although there is unused concurrency elsewhere in your account, function-orange cannot access it. The red line indicates that function-orange is experiencing throttling, and Lambda may drop requests.

• At t4, function-orange starts to receive fewer requests and is no longer throttling. However, your other functions experience a spike in traffic and begin throttling. Although there is unused concurrency elsewhere in your account, these other functions cannot access it. The red line indicates that your other functions are experiencing throttling.

• At t5, other functions start to receive fewer requests and are no longer throttling.

From this example, notice that reserving concurrency has the following effects:

• **Your function can scale independently of other functions in your account.** All of your account's functions in the same Region that don't have reserved concurrency share the pool of unreserved concurrency. Without reserved concurrency, other functions can potentially use up all of your available concurrency. This prevents critical functions from scaling up if needed.

• **Your function can't scale out of control.** Reserved concurrency caps your function's maximum concurrency. This means that your function can't use concurrency reserved for other functions, or concurrency from the unreserved pool. You can reserve concurrency to prevent your function from using all the available concurrency in your account, or from overloading downstream resources.

• **You may not be able to use all of your account's available concurrency.** Reserving concurrency counts towards your account concurrency limit, but this also means that other functions cannot use that chunk of reserved concurrency. If your function doesn't use up all of the concurrency that you reserve for it, you're effectively wasting that concurrency. This isn't an issue unless other functions in your account could benefit from the wasted concurrency.
To learn how to manage reserved concurrency settings for your functions, see Configuring reserved concurrency.

Provisioned concurrency

You use reserved concurrency to define the maximum number of execution environments reserved for a Lambda function. However, none of these environments come pre-initialized. As a result, your function invocations may take longer because Lambda must first initialize the new environment before being able to use it to invoke your function. When Lambda has to initialize a new environment in order to carry out an invocation, this is known as a cold start. To mitigate cold starts, you can use provisioned concurrency.

Provisioned concurrency is the number of pre-initialized execution environments that you want to allocate to your function. If you set provisioned concurrency on a function, Lambda initializes that number of execution environments so that they are prepared to respond immediately to function requests.

Note

Using provisioned concurrency incurs additional charges to your account. If you're working with the Java 11 or Java 17 runtimes, you can also use Lambda SnapStart to mitigate cold start issues at no additional cost. SnapStart uses cached snapshots of your execution environment to significantly improve startup performance. You cannot use both SnapStart and provisioned concurrency on the same function version. For more information about SnapStart features, limitations, and supported Regions, see Improving startup performance with Lambda SnapStart.

When using provisioned concurrency, Lambda still recycles execution environments in the background. However, at any given time, Lambda always ensures that the number of pre-initialized environments is equal to the value of your function's provisioned concurrency setting. This behavior differs from reserved concurrency, where Lambda may completely terminate an environment after a period of inactivity. The following diagram illustrates this by comparing the lifecycle of a single execution environment when you configure your function using reserved concurrency compared to provisioned concurrency.
The diagram has four points of interest:

<table>
<thead>
<tr>
<th>Time</th>
<th>Reserved concurrency</th>
<th>Provisioned concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Nothing happens.</td>
<td>Lambda pre-initializes an execution environment instance.</td>
</tr>
<tr>
<td>t2</td>
<td>Request 1 comes in. Lambda must initialize a new execution environment instance.</td>
<td>Request 1 comes in. Lambda uses the pre-initialized environment instance.</td>
</tr>
<tr>
<td>t3</td>
<td>After some inactivity, Lambda terminates the active environment instance.</td>
<td>Nothing happens.</td>
</tr>
<tr>
<td>t4</td>
<td>Request 2 comes in. Lambda must initialize a new execution environment instance.</td>
<td>Request 2 comes in. Lambda uses the pre-initialized environment instance.</td>
</tr>
</tbody>
</table>

To better understand provisioned concurrency, consider the following diagram:
In this diagram, you have an account concurrency limit of 1,000. You decide to give 400 units of provisioned concurrency to function-orange. All functions in your account, including function-orange, can use the remaining 600 units of unreserved concurrency.

The diagram has five points of interest:

- At t1, function-orange begins receiving requests. Since Lambda has pre-initialized 400 execution environment instances, function-orange is ready for immediate invocation.
- At t2, function-orange reaches 400 concurrent requests. As a result, function-orange runs out of provisioned concurrency. However, since there's still unreserved concurrency available, Lambda can use this to handle additional requests to function-orange (there's no throttling). Lambda must create new instances to serve these requests, and your function may experience cold start latencies.
- At t3, function-orange returns to 400 concurrent requests after a brief spike in traffic. Lambda is again able to handle all requests without cold start latencies.
- At t4, functions in your account experience a burst in traffic. This burst can come from function-orange or any other function in your account. Lambda uses unreserved concurrency to handle these requests.
At t5, functions in your account reach the maximum concurrency limit of 1,000, and experience throttling.

The previous example considered only provisioned concurrency. In practice, you can set both provisioned concurrency and reserved concurrency on a function. You might do this if you had a function that handles a consistent load of invocations on weekdays, but routinely sees spikes of traffic on weekends. In this case, you could use provisioned concurrency to set a baseline amount of environments to handle request during weekdays, and use reserved concurrency to handle the weekend spikes. Consider the following diagram:

In this diagram, suppose that you configure 200 units of provisioned concurrency and 400 units of reserved concurrency for function-orange. Because you configured reserved concurrency, function-orange cannot use any of the 600 units of unreserved concurrency.

This diagram has five points of interest:

- At t1, function-orange begins receiving requests. Since Lambda has pre-initialized 200 execution environment instances, function-orange is ready for immediate invocation.
- At t2, function-orange uses up all its provisioned concurrency. function-orange can continue serving requests using reserved concurrency, but these requests may experience cold start latencies.
• At t3, function-orange reaches 400 concurrent requests. As a result, function-orange uses up all its reserved concurrency. Since function-orange cannot use unreserved concurrency, requests begin to throttle.

• At t4, function-orange starts to receive fewer requests, and no longer throttles.

• At t5, function-orange drops down to 200 concurrent requests, so all requests are again able to use provisioned concurrency (that is, no cold start latencies).

Both reserved concurrency and provisioned concurrency count towards your account concurrency limit and Regional quotas. In other words, allocating reserved and provisioned concurrency can impact the concurrency pool that's available to other functions. Configuring provisioned concurrency incurs charges to your AWS account.

Note

If the amount of provisioned concurrency on a function's versions and aliases adds up to the function's reserved concurrency, then all invocations run on provisioned concurrency. This configuration also has the effect of throttling the unpublished version of the function ($LATEST), which prevents it from executing. You can't allocate more provisioned concurrency than reserved concurrency for a function.

To manage provisioned concurrency settings for your functions, see Configuring provisioned concurrency. To automate provisioned concurrency scaling based on a schedule or application utilization, see Managing provisioned concurrency with Application Auto Scaling.

How Lambda allocates provisioned concurrency

Provisioned concurrency doesn't come online immediately after you configure it. Lambda starts allocating provisioned concurrency after a minute or two of preparation. In particular, Lambda can provision between 500 to 3,000 execution environments at once, depending on the Region. After this initial burst, Lambda allocates 500 additional environments per minute, regardless of the Region, until the request is fulfilled.

For example, suppose your account concurrency limit is 10,000. Also, suppose that at 10:00 in US East (N. Virginia), you configure 5,000 units of provisioned concurrency for one function. Here's how Lambda might allocate provisioned concurrency units:
In the previous diagram:

- Initially, Lambda can provision at max 3,000 execution environments, since the initial burst concurrency limit in US East (N. Virginia) is 3,000.
- **At 10:00**: You request 5,000 units of provisioned concurrency for this function. Lambda doesn't begin provisioning execution environments instantaneously.
- **At 10:01**: Lambda starts by provisioning 3,000 environments.
- **From 10:02 to 10:05**: Lambda provisions 500 additional environments each minute. By 10:05, Lambda finishes allocating 5,000 environments to your function.

When you submit a request to allocate provisioned concurrency, you can't access any of those environments until Lambda completely finishes allocating them. For example, in the previous scenario, none of your requests can use provisioned concurrency until 10:05, since that's when Lambda completely finishes allocating your request of 5,000 execution environments.

**Comparing reserved concurrency and provisioned concurrency**

The following table summarizes and compares reserved and provisioned concurrency.

<table>
<thead>
<tr>
<th>Scaling quota available</th>
<th>Provisioned concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,000</td>
<td>0</td>
</tr>
<tr>
<td>3,000</td>
<td>3,000</td>
</tr>
<tr>
<td>500</td>
<td>3,500</td>
</tr>
<tr>
<td>500</td>
<td>4,000</td>
</tr>
<tr>
<td>500</td>
<td>4,500</td>
</tr>
<tr>
<td>500</td>
<td>5,000</td>
</tr>
<tr>
<td>1,000</td>
<td>5,000</td>
</tr>
<tr>
<td>Topic</td>
<td>Reserved concurrency</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Definition</td>
<td>Maximum number of execution environment instances for your function.</td>
</tr>
<tr>
<td>Provisioning behavior</td>
<td>Lambda provisions new instances on an on-demand basis.</td>
</tr>
<tr>
<td>Cold start behavior</td>
<td>Cold start latency possible, since Lambda must create new instances on-demand.</td>
</tr>
<tr>
<td>Throttling behavior</td>
<td>Function throttled when reserved concurrency limit reached.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Default behavior if not set</td>
<td>Function uses unreserved concurrency available in your account.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Pricing</td>
<td>No additional charge.</td>
</tr>
</tbody>
</table>
Concurrency quotas

Lambda sets quotas for the total amount of concurrency that you can use across all functions in a Region. These quotas exist on two levels:

- **At the account level**, your functions can have up to 1,000 units of concurrency by default. To increase this limit, see [Requesting a quota increase](https://docs.aws.amazon.com/service-quotas/latest/userguide/service-quotas-aws-lambda.html) in the *Service Quotas User Guide*.

- **At the function level**, you can reserve up to 900 units of concurrency across all your functions by default. Regardless of your total account concurrency limit, Lambda always reserves 100 units of concurrency for your functions that don't explicitly reserve concurrency. For example, if you increased your account concurrency limit to 2,000, then you can reserve up to 1,900 units of concurrency at the function level.

To check your current account level concurrency quota, use the AWS Command Line Interface (AWS CLI) to run the following command:

```
aws lambda get-account-settings
```

You should see output that looks like the following:

```
{
   "AccountLimit": {
       "TotalCodeSize": 80530636800,
       "CodeSizeUnzipped": 262144000,
       "CodeSizeZipped": 52428800,
       "ConcurrentExecutions": 1000,
       "UnreservedConcurrentExecutions": 900
   },
   "AccountUsage": {
       "TotalCodeSize": 410759889,
       "FunctionCount": 8
   }
}
```

`ConcurrentExecutions` is your total account-level concurrency quota. `UnreservedConcurrentExecutions` is the amount of reserved concurrency that you can still allocate to your functions.
As your function receives more requests, Lambda automatically scales up the number of execution environments to handle these requests until your account reaches its concurrency quota. However, to protect against over-scaling in response to sudden bursts of traffic, Lambda limits how fast your functions can scale. This scaling rate is the maximum rate at which functions in your account can scale in response to increased requests. (That is, how quickly Lambda can create new execution environments.)

In each AWS Region, and for each function, your concurrency scaling rate is **1,000 execution environment instances every 10 seconds**. In other words, every 10 seconds, Lambda can allocate at most 1,000 additional execution environment instances to each of your functions.

Usually, you don't need to worry about this limitation. Lambda's scaling rate is sufficient for most use cases.

Importantly, the concurrency scaling limit is a function-level limit. This means that each function in your account can scale independently of other functions.

For more information about scaling behavior, see [Lambda scaling behavior](#).
Configuring reserved concurrency

In Lambda, concurrency is the number of in-flight requests that your function is handling at the same time. There are two types of concurrency controls available:

- **Reserved concurrency** – Reserved concurrency is the maximum number of concurrent instances that you want to allocate to your function. When a function has reserved concurrency, no other function can use that concurrency. There is no charge for configuring reserved concurrency for a function.

- **Provisioned concurrency** – Provisioned concurrency is the number of pre-initialized execution environments that you want to allocate to your function. These execution environments are prepared to respond immediately to incoming function requests. Configuring provisioned concurrency incurs charges to your AWS account.

This topic details how to manage and configure reserved concurrency. For a conceptual overview of these two types of concurrency controls, see [Reserved concurrency and provisioned concurrency](#). For information on configuring provisioned concurrency, see the section called “Configuring provisioned concurrency”.

**Note**

Lambda functions that an Amazon MQ event source mapping invokes have a default maximum concurrency. For Apache Active MQ, the maximum number of concurrent instances is 5. For Rabbit MQ, the maximum number of concurrent instances is 1. Setting reserved or provisioned concurrency for your function doesn't change these limits. To request an increase in the default maximum concurrency when using Amazon MQ, contact AWS Support.

**Sections**

- [Configuring reserved concurrency](#)
- [Configuring concurrency with the Lambda API](#)

**Configuring reserved concurrency**

You can configure reserved concurrency settings for a function using the Lambda console or the Lambda API.
To reserve concurrency for a function (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function you want to reserve concurrency for.
3. Choose **Configuration** and then choose **Concurrency**.
4. Under **Concurrency**, choose **Edit**.
5. Choose **Reserve concurrency**. Enter the amount of concurrency to reserve for the function.
6. Choose **Save**.

You can reserve up to the **Unreserved account concurrency** value minus 100. The remaining 100 units of concurrency are for functions that aren’t using reserved concurrency. For example, if your account has a concurrency limit of 1,000, you cannot reserve all 1,000 units of concurrency to a single function.

### Edit concurrency

<table>
<thead>
<tr>
<th>Concurrency</th>
</tr>
</thead>
</table>

Unreserved account concurrency: 0

- Use unreserved account concurrency
- Reserve concurrency

1000

⚠️ The unreserved account concurrency can't go below 100.

Reserving concurrency for a function impacts the concurrency pool that’s available to other functions. For example, if you reserve 100 units of concurrency for function-a, other functions in your account must share the remaining 900 units of concurrency, even if function-a doesn't use all 100 reserved concurrency units.

To intentionally throttle a function, set its reserved concurrency to 0. This stops your function from processing any events until you remove the limit.
To configure reserved concurrency with the Lambda API, use the following API operations.

- **PutFunctionConcurrency**
- **GetFunctionConcurrency**
- **DeleteFunctionConcurrency**

For example, to configure reserved concurrency with the AWS Command Line Interface (CLI), use the `put-function-concurrency` command. The following command reserves 100 concurrency units for a function named `my-function`:

```bash/aws lambda put-function-concurrency --function-name my-function \\ --reserved-concurrent-executions 100```

You should see output that looks like the following:

```json
{
   "ReservedConcurrentExecutions": 100
}
```

**Configuring concurrency with the Lambda API**

If your function is currently serving traffic, you can easily view its concurrency metrics using CloudWatch metrics. Specifically, the `ConcurrentExecutions` metric shows you the number of concurrent invocations for each function in your account.

The previous graph suggests that this function serves an average of 5 to 10 concurrent requests at any given time, and peaks at 20 requests on a typical day. Suppose that there are many other functions in your account. **If this function is critical to your application and you don't want to drop any requests**, use a number greater than or equal to 20 as your reserved concurrency setting.
Alternatively, recall that you can also calculate concurrency using the following formula:

\[
\text{Concurrency} = (\text{average requests per second}) \times (\text{average request duration in seconds})
\]

Multiplying average requests per second with the average request duration in seconds gives you a rough estimate of how much concurrency you need to reserve. You can estimate average requests per second using the Invocation metric, and the average request duration in seconds using the Duration metric. See Working with Lambda function metrics for more details.
Configuring provisioned concurrency

In Lambda, concurrency is the number of in-flight requests that your function is handling at the same time. There are two types of concurrency controls available:

- **Reserved concurrency** – Reserved concurrency is the maximum number of concurrent instances that you want to allocate to your function. When a function has reserved concurrency, no other function can use that concurrency. There is no charge for configuring reserved concurrency for a function.

- **Provisioned concurrency** – Provisioned concurrency is the number of pre-initialized execution environments that you want to allocate to your function. These execution environments are prepared to respond immediately to incoming function requests. Configuring provisioned concurrency incurs charges to your AWS account.

This topic details how to manage and configure provisioned concurrency. For a conceptual overview of these two types of concurrency controls, see [Reserved concurrency and provisioned concurrency](#). For more information on configuring reserved concurrency, see [the section called “Configuring reserved concurrency”](#).

### Note

Lambda functions that an Amazon MQ event source mapping invokes have a default maximum concurrency. For Apache Active MQ, the maximum number of concurrent instances is 5. For Rabbit MQ, the maximum number of concurrent instances is 1. Setting reserved or provisioned concurrency for your function doesn't change these limits. To request an increase in the default maximum concurrency when using Amazon MQ, contact AWS Support.

**Sections**

- [Configuring provisioned concurrency](#)
- [Accurately estimating required provisioned concurrency](#)
- [Optimizing latency with provisioned concurrency](#)
- [Managing provisioned concurrency with Application Auto Scaling](#)
Configuring provisioned concurrency

You can configure provisioned concurrency settings for a function using the Lambda console or the Lambda API.

To allocate provisioned concurrency for a function (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function you want to allocate provisioned concurrency for.
3. Choose **Configuration** and then choose **Concurrency**.
4. Under **Provisioned concurrency configurations**, choose **Add configuration**.
5. Choose **Reserve concurrency**. Enter the amount of concurrency to reserve for the function.
6. Choose the qualifier type, and alias or version.

⚠️ **Note**

You cannot use provisioned concurrency with the $LATEST version of any function. In addition, if you're using an event source with your Lambda function, make sure that event source points to the correct alias or version. Otherwise, your function won't use provisioned concurrency environments.

7. Enter a number under **Provisioned concurrency**. Lambda provides an estimate of monthly costs.
8. Choose **Save**.

You can configure up to the **Unreserved account concurrency** in your account, minus 100. The remaining 100 units of concurrency are for functions that aren't using reserved concurrency. For example, if your account has a concurrency limit of 1,000, and you haven't assigned any reserved or provisioned concurrency to any of your other functions, you can configure a maximum of 900 provisioned concurrency units for a single function.
Configuring provisioned concurrency for a function impacts the concurrency pool that's available to other functions. For example, if you configure 100 units of provisioned concurrency for function-a, other functions in your account must share the remaining 900 units of concurrency, even if function-a doesn't use all 100 provisioned concurrency units.

You can allocate both reserved concurrency and provisioned concurrency for the same function. If you do so, the amount of provisioned concurrency cannot exceed the amount of reserved concurrency.

This limit also applies to function versions. The maximum amount of provisioned concurrency you can allocate to a specific function version is equal to the function's reserved concurrency minus the provisioned concurrency on other function versions.

To configure provisioned concurrency with the Lambda API, use the following API operations.

- **PutProvisionedConcurrencyConfig**
- **GetProvisionedConcurrencyConfig**
- **ListProvisionedConcurrencyConfigs**
- **DeleteProvisionedConcurrencyConfig**

For example, to configure provisioned concurrency with the AWS Command Line Interface (CLI), use the `put-provisioned-concurrency-config` command. The following command allocates 100 units of provisioned concurrency for the BLUE alias of a function named `my-function`:

```
aws lambda put-provisioned-concurrency-config --function-name my-function \
```

---

Provisioned concurrency
To enable your function to scale without fluctuations in latency, use provisioned concurrency. You can use Application Auto Scaling to automatically adjust provisioned concurrency to maintain a configured target utilization. Provisioned concurrency runs continually and has separate pricing for concurrency and execution duration. [Learn more](#)

$0.00 per month in addition to pricing for duration and requests. [Pricing](#)

<table>
<thead>
<tr>
<th>Units of Provisioned Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
</tr>
</tbody>
</table>

⚠️ The maximum allowed provisioned concurrency is 900, based on the unreserved concurrency available (1000) minus the minimum unreserved account concurrency (100).
900 available

Please correct the errors above.
--qualifier BLUE \
--provisioned-concurrent-executions 100

You should see output that looks like the following:

```json
{
  "Requested ProvisionedConcurrentExecutions": 100,
  "Allocated ProvisionedConcurrentExecutions": 0,
  "Status": "IN_PROGRESS",
  "LastModified": "2023-01-21T11:30:00+0000"
}
```

**Accurately estimating required provisioned concurrency**

If your function is currently serving traffic, you can easily view its concurrency metrics using [CloudWatch metrics](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/cloudwatch-overview-whatis.html). Specifically, the **ConcurrentExecutions** metric shows you the number of concurrent invocations for each function in your account.

The previous graph suggests that this function serves an average of 5 to 10 concurrent requests at any given time, and peaks at 20 requests on a typical day. Suppose that there are many other functions in your account. **If this function is critical to your application and you need a low-latency response on every invocation**, use a number greater than or equal to 20 as your provisioned concurrency setting.

Alternatively, recall that you can also **calculate concurrency** using the following formula:

Concurrency = (average requests per second) * (average request duration in seconds)

Multiplying average requests per second with the average request duration in seconds gives you a rough estimate of how much concurrency you need to reserve. You can estimate average requests
per second using the Invocation metric, and the average request duration in seconds using the Duration metric. See Working with Lambda function metrics for more details.

When working with provisioned concurrency, Lambda suggests including a 10% buffer on top of the amount of concurrency your function typically needs. For example, if your function usually peaks at 200 concurrent requests, set your provisioned concurrency at 220 instead (200 concurrent requests + 10% = 220 provisioned concurrency).

**Optimizing latency with provisioned concurrency**

The way you structure your function code to optimize for latency can depend on whether you choose provisioned concurrency or on-demand environments. For functions running on provisioned concurrency, Lambda runs any initialization code (i.e. loading libraries and instantiating clients) at allocation time. So, putting as much initialization outside of the main function handler is a good idea, since doing so won't impact latency during actual function invocations. In contrast, if you initialize libraries or instantiate clients within your main handler code, your function has to run this each time you invoke it, regardless of whether or not you're using provisioned concurrency.

If you're using on-demand instances, Lambda may have to re-run your initialization code every time your function receives a request (cold start). Depending on what your function needs to achieve, you may choose to defer initialization for a specific capability until the function needs that capability. For example, consider the following control flow for a Lambda handler:

```python
def handler(event, context):
    ...
    if ( some_condition ):
        // Initialize CLIENT_A to perform a task
    else:
        // Do nothing
```

In the previous example, instead of initializing CLIENT_A outside of the main handler, the function author chose to initialize it within the if statement. By doing this, Lambda only runs this code if some_condition is satisfied. If the author initializes CLIENT_A outside the main handler, Lambda runs that code on every cold start, increasing overall latency.

It's possible for your function to use up all of its provisioned concurrency. To handle excess traffic, your function has to use on-demand instances. To help you determine what type of initialization Lambda used for a particular environment, check the value of the AWS_LAMBDA_INITIALIZATION_TYPE environment variable. This variable can have two possible values: provisioned-concurrency or on-demand. The value of
AWS_LAMBDA_INITIALIZATION_TYPE is immutable and does not change over the lifetime of the execution environment.

If you use the .NET 6 or .NET 7 runtimes, you can configure the AWS_LAMBDA_DOTNET_PREJIT environment variable to improve the latency for functions, even if they don't use provisioned concurrency. The .NET runtime lazily compiles and initializes each library that your code calls for the first time. As a result, the first invocation of a Lambda function can take longer than subsequent invocations. To mitigate this, you can choose one of three values for AWS_LAMBDA_DOTNET_PREJIT:

- **ProvisionedConcurrency**: Lambda performs ahead-of-time JIT compilation for all environments using provisioned concurrency. This is the default value.
- **Always**: Lambda performs ahead-of-time JIT compilation for every environment, even if the function doesn't use provisioned concurrency.
- **Never**: Lambda disables ahead-of-time JIT compilation for all environments.

For provisioned concurrency environments, your function's initialization code runs during allocation, and every few hours as Lambda recycles active instances of your environment. You can see the initialization time in logs and traces after an environment instance processes a request. However, Lambda bills you for initialization even if the environment instance never processes a request. Provisioned concurrency runs continually and is billed separately from initialization and invocation costs. For details, see AWS Lambda Pricing.

For additional guidance on optimizing functions using provisioned concurrency, see Lambda execution environments in Serverless Land.

### Managing provisioned concurrency with Application Auto Scaling

You can use Application Auto Scaling to manage provisioned concurrency on a schedule or based on utilization. If you observe predictable patterns of traffic to your function, use scheduled scaling. If you want your function to maintain a specific utilization percentage, use a target tracking scaling policy.

#### Scheduled scaling

With Application Auto Scaling, you can set your own scaling schedule according to predictable load changes. For more information and examples, see Scheduled scaling for Application Auto Scaling in the Application Auto Scaling User Guide, and Scheduling AWS Lambda Provisioned Concurrency for recurring peak usage on the AWS Compute Blog.
Target tracking

With target tracking, Application Auto Scaling creates and manages a set of CloudWatch alarms based on how you define your scaling policy. When these alarms activate, Application Auto Scaling automatically adjusts the amount of environments allocated using provisioned concurrency. Target tracking is ideal for applications that don't have predictable traffic patterns.

To scale provisioned concurrency using target tracking, use the RegisterScalableTarget and PutScalingPolicy Application Auto Scaling API operations. For example, if you're using the AWS Command Line Interface (CLI), follow these steps:

1. Register a function's alias as a scaling target. The following example registers the BLUE alias of a function named my-function:

   ```bash
   aws application-autoscaling register-scalable-target --service-namespace lambda \
   --resource-id function:my-function:BLUE --min-capacity 1 --max-capacity 100 \
   --scalable-dimension lambda:function:ProvisionedConcurrency
   ```

2. Apply a scaling policy to the target. The following example configures Application Auto Scaling to adjust the provisioned concurrency configuration for an alias to keep utilization near 70 percent.

   ```bash
   aws application-autoscaling put-scaling-policy \
   --service-namespace lambda \
   --scalable-dimension lambda:function:ProvisionedConcurrency \
   --resource-id function:my-function:BLUE \
   --policy-name my-policy \
   --policy-type TargetTrackingScaling \
   --target-tracking-scaling-policy-configuration '{ "TargetValue": 0.7, "PredefinedMetricSpecification": { "PredefinedMetricType": "LambdaProvisionedConcurrencyUtilization" }}'
   ```

You should see output that looks like this:

```json
{
   "Alarms": [
      {}
   ]
}
```
Application Auto Scaling creates two alarms in CloudWatch. The first alarm triggers when the utilization of provisioned concurrency consistently exceeds 70%. When this happens, Application Auto Scaling allocates more provisioned concurrency to reduce utilization. The second alarm triggers when utilization is consistently less than 63% (90 percent of the 70% target). When this happens, Application Auto Scaling reduces the alias's provisioned concurrency.

In the following example, a function scales between a minimum and maximum amount of provisioned concurrency based on utilization.
When the number of open requests increase, Application Auto Scaling increases provisioned concurrency in large steps until it reaches the configured maximum. Once it reaches the maximum, the function can continue to scale on standard, unreserved concurrency if your account hasn't reached its account concurrency limit. When utilization drops and stays consistently low, Application Auto Scaling decreases provisioned concurrency in smaller periodic steps.
Both of the alarms that Application Auto Scaling manages use the average statistic by default. Functions that have traffic patterns that come in quick bursts may not trigger these alarms. For example, suppose your Lambda function executes quickly (i.e. 20-100 ms) and your traffic pattern comes in quick bursts. In this case, the number of requests may exceed allocated provisioned concurrency during the burst, but the burst load must sustain for at least 3 minutes for Application Auto Scaling to provision additional environments. Additionally, both CloudWatch alarms require 3 data points that hit the target average before activating the auto scaling policy.

For more information on target tracking scaling policies, see Target tracking scaling policies for Application Auto Scaling.
**Lambda scaling behavior**

As your function receives more requests, Lambda automatically scales up the number of execution environments to handle these requests until your account reaches its concurrency quota. However, to protect against over-scaling in response to sudden bursts of traffic, Lambda limits how fast your functions can scale. This scaling rate is the maximum rate at which functions in your account can scale in response to increased requests. (That is, how quickly Lambda can create new execution environments.)

**Concurrency scaling rate**

*In each AWS Region, and for each function, your concurrency scaling rate is 1,000 execution environment instances every 10 seconds.* In other words, every 10 seconds, Lambda can allocate at most 1,000 additional execution environment instances to each of your functions.

Usually, you don't need to worry about this limitation. Lambda's scaling rate is sufficient for most use cases.

Importantly, the concurrency scaling limit is a function-level limit. This means that each function in your account can scale independently of other functions.

⚠️ **Note**

In practice, Lambda makes a best attempt to refill your concurrency scaling rate continuously over time, rather than in one single refill of 1,000 units every 10 seconds.

Lambda doesn't accrue unused portions of your concurrency scaling rate. This means that at any instant in time, your scaling rate is always 1,000 concurrency units at maximum. For example, if you don't use any of your available 1,000 concurrency units in a 10-second interval, you won't accrue 1,000 additional units in the next 10-second interval. Your concurrency scaling rate is still 1,000 in the next 10-second interval.

As long as your function continues to receive increasing numbers of requests, then Lambda scales at the fastest rate available to you, up to your account's concurrency limit. You can limit the amount of concurrency that individual functions can use by configuring reserved concurrency. If requests come in faster than your function can scale, or if your function is at maximum concurrency, then additional requests fail with a throttling error (429 status code).
Monitoring concurrency

Lambda emits Amazon CloudWatch metrics to help you monitor concurrency for your functions. This topic explains these metrics and how to interpret them.

Sections

- General concurrency metrics
- Provisioned concurrency metrics
- Working with the ClaimedAccountConcurrency metric

General concurrency metrics

Use the following metrics to monitor concurrency for your Lambda functions. The granularity for each metric is 1 minute.

- ConcurrentExecutions – The number of active concurrent invocations at a given point in time. Lambda emits this metric for all functions, versions, and aliases. For any function in the Lambda console, Lambda displays the graph for ConcurrentExecutions natively in the Monitoring tab, under Metrics. View this metric using MAX.

- UnreservedConcurrentExecutions – The number of active concurrent invocations that are using unreserved concurrency. Lambda emits this metric across all functions in a region. View this metric using MAX.

- ClaimedAccountConcurrency – The amount of concurrency that is unavailable for on-demand invocations. ClaimedAccountConcurrency is equal to UnreservedConcurrentExecutions plus the amount of allocated concurrency (i.e. the total reserved concurrency plus total provisioned concurrency). If ClaimedAccountConcurrency exceeds your account concurrency limit, you can request a higher account concurrency limit. View this metric using MAX. For more information, see Working with the ClaimedAccountConcurrency metric.

Provisioned concurrency metrics

Use the following metrics to monitor Lambda functions using provisioned concurrency. The granularity for each metric is 1 minute.
• **ProvisionedConcurrentExecutions** – The number of execution environment instances that are actively processing an invocation on provisioned concurrency. Lambda emits this metric for each function version and alias with provisioned concurrency configured. View this metric using \textbf{MAX}.

ProvisionedConcurrentExecutions is not the same as the total number of provisioned concurrency that you allocate. For example, suppose you allocate 100 units of provisioned concurrency to a function version. During any given minute, if at most 50 out of those 100 execution environments were handling invocations simultaneously, then the value of \textbf{MAX}(ProvisionedConcurrentExecutions) is 50.

• **ProvisionedConcurrentInvocations** – The number of times Lambda invokes your function code using provisioned concurrency. Lambda emits this metric for each function version and alias with provisioned concurrency configured. View this metric using \textbf{SUM}.

ProvisionedConcurrentInvocations differs from ProvisionedConcurrentExecutions in that ProvisionedConcurrentInvocations counts total number of invocations, while ProvisionedConcurrentExecutions counts number of active environments. To understand this distinction, consider the following scenario:

In this example, suppose that you receive 1 invocation per minute, and each invocation takes 2 minutes to complete. Each orange horizontal bar represents a single request. Suppose that you allocate 10 units of provisioned concurrency to this function, such that each request runs on provisioned concurrency.
In between minutes 0 and 1, Request 1 comes in. At minute 1, the value for \( \text{MAX(ProvisionedConcurrentExecutions)} \) is 1, since at most 1 execution environment was active during the past minute. The value for \( \text{SUM(ProvisionedConcurrentInvocations)} \) is also 1, since 1 new request came in during the past minute.

In between minutes 1 and 2, Request 2 comes in, and Request 1 continues to run. At minute 2, the value for \( \text{MAX(ProvisionedConcurrentExecutions)} \) is 2, since at most 2 execution environments were active during the past minute. However, the value for \( \text{SUM(ProvisionedConcurrentInvocations)} \) is 1, since only 1 new request came in during the past minute. This metric behavior continues until the end of the example.

- **ProvisionedConcurrencySpillOverInvocations** – The number of times Lambda invokes your function on standard (reserved or unreserved) concurrency when all provisioned concurrency is in use. Lambda emits this metric for each function version and alias with provisioned concurrency configured. View this metric using \( \text{SUM} \). The value of \( \text{ProvisionedConcurrentInvocations} + \text{ProvisionedConcurrencySpillOverInvocations} \) should be equal to the total number of function invocations (i.e. the Invocations metric).

- **ProvisionedConcurrencyUtilization** – The percentage of provisioned concurrency in use (i.e. the value of \( \text{ProvisionedConcurrentExecutions} \) divided by the total amount of provisioned concurrency allocated). Lambda emits this metric for each function version and alias with provisioned concurrency configured. View this metric using \( \text{MAX} \).

For example, suppose you provision 100 units of provisioned concurrency to a function version. During any given minute, if at most 60 out of those 100 execution environments were handling invocations simultaneously, then the value of \( \text{MAX(ProvisionedConcurrentExecutions)} \) is 60, and the value of \( \text{MAX(ProvisionedConcurrencyUtilization)} \) is 0.6.

A high value for \( \text{ProvisionedConcurrencySpillOverInvocations} \) may indicate that you need to allocate additional provisioned concurrency for your function. Alternatively, you can configure Application Auto Scaling to handle automatic scaling of provisioned concurrency based on pre-defined thresholds.

Conversely, consistently low values for \( \text{ProvisionedConcurrencyUtilization} \) may indicate that you over-allocated provisioned concurrency for your function.
Working with the ClaimedAccountConcurrency metric

Lambda uses the ClaimedAccountConcurrency metric to determine how much concurrency your account is available for on-demand invocations. Lambda calculates ClaimedAccountConcurrency using the following formula:

\[
\text{ClaimedAccountConcurrency} = \text{UnreservedConcurrentExecutions} + \text{(allocated concurrency)}
\]

UnreservedConcurrentExecutions is the number of active concurrent invocations that are using unreserved concurrency. Allocated concurrency is the sum of the following two parts (substituting RC as "reserved concurrency" and PC as "provisioned concurrency"):

- The total RC across all functions in a Region.
- The total PC across all functions in a Region that use PC, excluding functions that use RC.

**Note**

You can't allocate more PC than RC for a function. Thus, a function's RC is always greater than or equal to its PC. To calculate the contribution to allocated concurrency for such functions with both PC and RC, Lambda considers only RC, which is the maximum of the two.

Lambda uses the ClaimedAccountConcurrency metric, rather than ConcurrentExecutions, to determine how much concurrency is available for on-demand invocations. While the ConcurrentExecutions metric is useful for tracking the number of active concurrent invocations, it doesn't always reflect your true concurrency availability. This is because Lambda also considers reserved concurrency and provisioned concurrency to determine availability.

To illustrate ClaimedAccountConcurrency, consider a scenario where you configure a lot of reserved concurrency and provisioned concurrency across your functions that go largely unused. In the following example, assume that your account concurrency limit is 1,000, and you have two main functions in your account: function-orange and function-blue. You allocate 600 units of reserved concurrency for function-orange. You allocate 200 units of provisioned concurrency for function-blue. Suppose that over time, you deploy additional functions and observe the following traffic pattern:
In the previous diagram, the black lines indicate the actual concurrency use over time, and the red line indicates the value of ClaimedAccountConcurrency over time. Throughout this scenario, ClaimedAccountConcurrency is 800 at minimum, despite low actual concurrency utilization across your functions. This is because you allocated 800 total units of concurrency for function-orange and function-blue. From Lambda's perspective, you have "claimed" this concurrency for use, so you effectively have only 200 units of concurrency remaining for other functions.

For this scenario, allocated concurrency is 800 in the ClaimedAccountConcurrency formula. We can then derive the value of ClaimedAccountConcurrency at various points in the diagram:

- At t1, ClaimedAccountConcurrency is 800 (800 + 0 UnreservedConcurrentExecutions).
- At t2, ClaimedAccountConcurrency is 900 (800 + 100 UnreservedConcurrentExecutions).
- At t3, ClaimedAccountConcurrency is again 900 (800 + 100 UnreservedConcurrentExecutions).
Setting up the `ClaimedAccountConcurrency` metric in CloudWatch

Lambda emits the `ClaimedAccountConcurrency` metric in CloudWatch. Use this metric along with the value of `SERVICE_QUOTA(ConcurrentExecutions)` to get the percent utilization of concurrency in your account, as shown in the following formula:

\[
\text{Utilization} = \left( \frac{\text{ClaimedAccountConcurrency}}{\text{SERVICE_QUOTA(ConcurrentExecutions)}} \right) \times 100\%
\]

The following screenshot illustrates how you can graph this formula in CloudWatch. The green `claim_utilization` line represents the concurrency utilization in this account, which is at around 40%:

The previous screenshot also includes a CloudWatch alarm that goes into ALARM state when the concurrency utilization exceeds 70%. You can use the `ClaimedAccountConcurrency` metric along with similar alarms to proactively determine when you might need to request a higher account concurrency limit.
Configuring code signing for AWS Lambda

Code signing for AWS Lambda helps to ensure that only trusted code runs in your Lambda functions. When you enable code signing for a function, Lambda checks every code deployment and verifies that the code package is signed by a trusted source.

Note
Functions defined as container images do not support code signing.

To verify code integrity, use AWS Signer to create digitally signed code packages for functions and layers. When a user attempts to deploy a code package, Lambda performs validation checks on the code package before accepting the deployment. Because code signing validation checks run at deployment time, there is no performance impact on function execution.

You also use AWS Signer to create signing profiles. You use a signing profile to create the signed code package. Use AWS Identity and Access Management (IAM) to control who can sign code packages and create signing profiles. For more information, see Authentication and Access Control in the AWS Signer Developer Guide.

To enable code signing for a function, you create a code signing configuration and attach it to the function. A code signing configuration defines a list of allowed signing profiles and the policy action to take if any of the validation checks fail.

Lambda layers follow the same signed code package format as function code packages. When you add a layer to a function that has code signing enabled, Lambda checks that the layer is signed by an allowed signing profile. When you enable code signing for a function, all layers that are added to the function must also be signed by one of the allowed signing profiles.

Use IAM to control who can create code signing configurations. Typically, you allow only specific administrative users to have this ability. Additionally, you can set up IAM policies to enforce that developers only create functions that have code signing enabled.

You can configure code signing to log changes to AWS CloudTrail. Successful and blocked deployments to functions are logged to CloudTrail with information about the signature and validation checks.
You can configure code signing for your functions using the Lambda console, the AWS Command Line Interface (AWS CLI), AWS CloudFormation, and the AWS Serverless Application Model (AWS SAM).

There is no additional charge for using AWS Signer or code signing for AWS Lambda.

**Sections**

- **Signature validation**
- **Configuration prerequisites**
- **Creating code signing configurations**
- **Updating a code signing configuration**
- **Deleting a code signing configuration**
- **Enabling code signing for a function**
- **Configuring IAM policies**
- **Configuring code signing with the Lambda API**

**Signature validation**

Lambda performs the following validation checks when you deploy a signed code package to your function:

1. **Integrity** – Validates that the code package has not been modified since it was signed. Lambda compares the hash of the package with the hash from the signature.
2. **Expiry** – Validates that the signature of the code package has not expired.
3. **Mismatch** – Validates that the code package is signed with one of the allowed signing profiles for the Lambda function. A mismatch also occurs if a signature is not present.
4. **Revocation** – Validates that the signature of the code package has not been revoked.

The signature validation policy defined in the code signing configuration determines which of the following actions Lambda takes if any of the validation checks fail:

- **Warn** – Lambda allows the deployment of the code package, but issues a warning. Lambda issues a new Amazon CloudWatch metric and also stores the warning in the CloudTrail log.
- **Enforce** – Lambda issues a warning (the same as for the Warn action) and blocks the deployment of the code package.
You can configure the policy for the expiry, mismatch, and revocation validation checks. Note that you cannot configure a policy for the integrity check. If the integrity check fails, Lambda blocks deployment.

Configuration prerequisites

Before you can configure code signing for a Lambda function, use AWS Signer to do the following:

- Create one or more signing profiles.
- Use a signing profile to create a signed code package for your function.

For more information, see Creating Signing Profiles (Console) in the AWS Signer Developer Guide.

Creating code signing configurations

A code signing configuration defines a list of allowed signing profiles and the signature validation policy.

To create a code signing configuration (console)

1. Open the Code signing configurations page of the Lambda console.
2. Choose Create configuration.
3. For Description, enter a descriptive name for the configuration.
4. Under Signing profiles, add up to 20 signing profiles to the configuration.
   a. For Signing profile version ARN, choose a profile version's Amazon Resource Name (ARN), or enter the ARN.
   b. To add an additional signing profile, choose Add signing profiles.
5. Under Signature validation policy, choose Warn or Enforce.
6. Choose Create configuration.

Updating a code signing configuration

When you update a code signing configuration, the changes impact the future deployments of functions that have the code signing configuration attached.
To update a code signing configuration (console)

1. Open the Code signing configurations page of the Lambda console.
2. Select a code signing configuration to update, and then choose Edit.
3. For Description, enter a descriptive name for the configuration.
4. Under Signing profiles, add up to 20 signing profiles to the configuration.
   a. For Signing profile version ARN, choose a profile version's Amazon Resource Name (ARN), or enter the ARN.
   b. To add an additional signing profile, choose Add signing profiles.
5. Under Signature validation policy, choose Warn or Enforce.
6. Choose Save changes.

Deleting a code signing configuration

You can delete a code signing configuration only if no functions are using it.

To delete a code signing configuration (console)

1. Open the Code signing configurations page of the Lambda console.
2. Select a code signing configuration to delete, and then choose Delete.
3. To confirm, choose Delete again.

Enabling code signing for a function

To enable code signing for a function, you associate a code signing configuration with the function.

To associate a code signing configuration with a function (console)

1. Open the Functions page of the Lambda console.
2. Choose the function for which you want to enable code signing.
4. In Edit code signing, choose a code signing configuration for this function.
5. Choose Save.
Configuring IAM policies

To grant permission for a user to access the code signing API operations, attach one or more policy statements to the user policy. For more information about user policies, see Identity-based IAM policies for Lambda.

The following example policy statement grants permission to create, update, and retrieve code signing configurations.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:CreateCodeSigningConfig",
        "lambda:UpdateCodeSigningConfig",
        "lambda:GetCodeSigningConfig"
      ],
      "Resource": "*"
    }
  ]
}
```

Administrators can use the CodeSigningConfigArn condition key to specify the code signing configurations that developers must use to create or update your functions.

The following example policy statement grants permission to create a function. The policy statement includes a lambda:CodeSigningConfigArn condition to specify the allowed code signing configuration. Lambda blocks any CreateFunction API request if its CodeSigningConfigArn parameter is missing or does not match the value in the condition.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowReferencingCodeSigningConfig",
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
      ],
    }
  ]
}
```
Configuring code signing with the Lambda API

To manage code signing configurations with the AWS CLI or AWS SDK, use the following API operations:

- **ListCodeSigningConfigs**
- **CreateCodeSigningConfig**
- **GetCodeSigningConfig**
- **UpdateCodeSigningConfig**
- **DeleteCodeSigningConfig**

To manage the code signing configuration for a function, use the following API operations:

- **CreateFunction**
- **GetFunctionCodeSigningConfig**
- **PutFunctionCodeSigningConfig**
- **DeleteFunctionCodeSigningConfig**
- **ListFunctionsByCodeSigningConfig**
Using tags on Lambda functions

You can tag AWS Lambda functions to activate attribute-based access control (ABAC) and to organize them by owner, project, or department. Tags are free-form key-value pairs that are supported across AWS services for use in ABAC, filtering resources, and adding detail to billing reports.

Tags apply at the function level, not to versions or aliases. Tags are not part of the version-specific configuration that Lambda creates a snapshot of when you publish a version.

Sections
- Permissions required for working with tags
- Using tags with the Lambda console
- Using tags with the AWS CLI
- Requirements for tags

Permissions required for working with tags

Grant appropriate permissions to the AWS Identity and Access Management (IAM) identity (user, group, or role) for the person working with the function:

- **lambda:ListTags** – When a function has tags, grant this permission to anyone who needs to call GetFunction or ListTags on it.
- **lambda:TagResource** – Grant this permission to anyone who needs to call CreateFunction or TagResource.

For more information, see Identity-based IAM policies for Lambda.

Using tags with the Lambda console

You can use the Lambda console to create functions that have tags, add tags to existing functions, and filter functions by tags that you add.

To add tags when you create a function

1. Open the Functions page of the Lambda console.
2. Choose **Create function**.
3. Choose **Author from scratch** or **Container image**.
4. Under **Basic information**, do the following:
   a. For **Function name**, enter the function name. Function names are limited to 64 characters in length.
   b. For **Runtime**, choose the language version to use for your function.
   c. (Optional) For **Architecture**, choose the instruction set architecture to use for your function. The default architecture is x86_64. When you build the deployment package for your function, make sure that it is compatible with the instruction set architecture that you choose.
5. Expand **Advanced settings**, and then select **Enable tags**.
6. Choose **Add new tag**, and then enter a **Key** and an optional **Value**. To add more tags, repeat this step.
7. Choose **Create function**.

**To add tags to an existing function**

1. Open the **Functions page** of the Lambda console.
2. Choose the name of a function.
3. Choose **Configuration**, and then choose **Tags**.
4. Under **Tags**, choose **Manage tags**.
5. Choose **Add new tag**, and then enter a **Key** and an optional **Value**. To add more tags, repeat this step.
6. Choose **Save**.
To filter functions with tags

1. Open the Functions page of the Lambda console.
2. Choose the search bar to see a list of function attributes and tag keys.

3. Choose a tag key to see a list of values that are in use in the current AWS Region.
4. Choose a value to see functions with that value, or choose (all values) to see all functions that have a tag with that key.

The search bar also supports searching for tag keys. Enter tag to see only a list of tag keys, or enter the name of a key to find it in the list.
Using tags with the AWS CLI

Adding and removing tags

To create a new Lambda function with tags, use the `create-function` command with the `--tags` option.

```bash
aws lambda create-function --function-name my-function
    --handler index.js --runtime nodejs20.x
    --role arn:aws:iam::123456789012:role/lambda-role
    --tags Department=Marketing,CostCenter=1234ABCD
```

To add tags to an existing function, use the `tag-resource` command.

```bash
aws lambda tag-resource
    --tags Department=Marketing,CostCenter=1234ABCD
```

To remove tags, use the `untag-resource` command.

```bash
aws lambda untag-resource
    --resource arn:aws:lambda:us-east-1:123456789012:function:my-function
    --tag-keys Department
```

Viewing tags on a function

If you want to view the tags that are applied to a specific Lambda function, you can use either of the following AWS CLI commands:

- **ListTags** – To view a list of the tags associated with this function, include your Lambda function ARN (Amazon Resource Name):

  ```bash
  aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:function:my-function
  ```

- **GetFunction** – To view a list of the tags associated with this function, include your Lambda function name:

  ```bash
  aws lambda get-function --function-name my-function
  ```
Filtering functions by tag

You can use the AWS Resource Groups Tagging API GetResources API operation to filter your resources by tags. The GetResources operation receives up to 10 filters, with each filter containing a tag key and up to 10 tag values. You provide GetResources with a ResourceType to filter by specific resource types.

For more information about AWS Resource Groups, see What are resource groups? in the AWS Resource Groups and Tags User Guide.

Requirements for tags

The following requirements apply to tags:

- Maximum number of tags per resource: 50
- Maximum key length: 128 Unicode characters in UTF-8
- Maximum value length: 256 Unicode characters in UTF-8
- Tag keys and values are case sensitive.
- Do not use the aws: prefix in your tag names or values because it is reserved for AWS use. You can't edit or delete tag names or values with this prefix. Tags with this prefix do not count against your tags per resource limit.
- If you plan to use your tagging schema across multiple services and resources, remember that other services may have restrictions on allowed characters. Generally allowed characters are: letters, spaces, and numbers representable in UTF-8, plus the following special characters: + - = . _ : / @.

Requirements for tags
Testing serverless functions and applications

Testing serverless functions uses traditional test types and techniques, but you must also consider testing serverless applications as a whole. Cloud-based tests will provide the most accurate measure of quality of both your functions and serverless applications.

A serverless application architecture includes managed services that provide critical application functionality through API calls. For this reason, your development cycle should include automated tests that verify functionality when your function and services interact.

If you do not create cloud-based tests, you could encounter issues due to differences between your local environment and the deployed environment. Your continuous integration process should run tests against a suite of resources provisioned in the cloud before promoting your code to the next deployment environment, such as QA, Staging, or Production.

Continue reading this short guide to learn about testing strategies for serverless applications, or visit the Serverless Test Samples repository to dive in with practical examples, specific to your chosen language and runtime.

For serverless testing, you will still write unit, integration and end-to-end tests.

• **Unit tests** - Tests that run against an isolated block of code. For example, verifying the business logic to calculate the delivery charge given a particular item and destination.
• **Integration tests** - Tests involving two or more components or services that interact, typically in a cloud environment. For example, verifying a function processes events from a queue.
End-to-end tests - Tests that verify behavior across an entire application. For example, ensuring infrastructure is set up correctly and that events flow between services as expected to record a customer's order.

Targeted business outcomes

Testing serverless solutions may require slightly more time to set up tests that verify event-driven interactions between services. Keep the following practical business reasons in mind as you read this guide:

- Increase the quality of your application
- Decrease time to build features and fix bugs

The quality of an application depends on testing a variety of scenarios to verify functionality. Carefully considering the business scenarios and automating those tests to run against cloud services will raise the quality of your application.

Software bugs and configuration problems have the least impact on cost and schedule when caught during an iterative development cycle. If issues remain undetected during development, finding and fixing in production requires more effort by more people.

A well planned serverless testing strategy will increase software quality and improve iteration time by verifying your Lambda functions and applications perform as expected in a cloud environment.

What to test

We recommend adopting a testing strategy that tests managed service behaviors, cloud configuration, security policies, and the integration with your code to improve software quality. Behavior testing, also known as black box testing, verifies a system works as expected without knowing all the internals.

- Run unit tests to check business logic inside Lambda functions.
- Verify integrated services are actually invoked, and input parameters are correct.
- Check that an event goes through all expected services end-to-end in a workflow.
In traditional server-based architecture, teams often define a scope for testing to only include code that runs on the application server. Other components, services, or dependencies are often considered external and out of scope for testing.

Serverless applications often consist of small units of work, such as Lambda functions that retrieve products from a database, or process items from a queue, or resize an image in storage. Each component runs in their own environment. Teams will likely be responsible for many of these small units within a single application.

Some application functionality can be delegated entirely to managed services such as Amazon S3, or created without using any internally developed code. There is no need to test these managed services, but you do need to test the integration with these services.

**How to test serverless**

You are probably familiar with how to test applications deployed locally: You write tests that run against code running entirely on your desktop operating system, or inside containers. For example, you might invoke a local web service component with a request and then make assertions about the response.

Serverless solutions are built from your function code and cloud-based managed services, such as queues, databases, event buses, and messaging systems. These components are all connected through an *event-driven architecture*, where messages, called *events*, flow from one resource to another. These interactions can be synchronous, such as when a web service returns results immediately, or an asynchronous action which complete at a later time, such as placing items in a queue or starting a workflow step. Your testing strategy must include both scenarios and test the interactions between services. For asynchronous interactions, you may need detect side effects in downstream components that may not be immediately observable.

Replicating an entire cloud environment, including queues, database tables, event buses, security policies, and more, is not practical. You will inevitably encounter issues due to differences between your local environment and your deployed environments in the cloud. The variations between your environments will increase the time to reproduce and fix bugs.

In serverless applications, architecture components commonly exist entirely in the cloud, so testing against code and services in the cloud is necessary to develop features and fix bugs.
Testing techniques

In reality, your testing strategy will likely include a mix of techniques to increase quality of your solutions. You will use quick interactive tests to debug functions in the console, automated unit tests to check isolated business logic, verification of calls to external services with mocks, and occasional testing against emulators that mimic a service.

- **Testing in the cloud** - you deploy infrastructure and code to test with actual services, security policies, configurations and infrastructure specific parameters. Cloud-based tests provide the most accurate measure of quality of your code.

  Debugging a function in the console is a quick way to test in the cloud. You can choose from a library of sample test events or create a custom event to test a function in isolation. You can also share test events through the console with your team.

  To automate testing in the development and build lifecycle, you will need to test outside of the console. See the language specific testing sections in this guide for automation strategies and resources.

- **Testing with mocks** (also called fakes) - Mocks are objects within your code that simulate and stand-in for an external service. Mocks provide pre-defined behavior to verify service calls and parameters. A fake is a mock implementation that takes shortcuts to simplify or improve performance. For example, a fake data access object might return data from an in-memory datastore. Mocks can mimic and simplify complex dependencies, but can also lead to more mocks in order to replace nested dependencies.

- **Testing with emulators** - You can setup applications (sometimes from a third party) to mimic a cloud service in your local environment. Speed is their strength, but setup and parity with production services is their weakness. Use emulators sparingly.

Testing in the cloud

Testing in the cloud is valuable for all phases of testing, including unit tests, integration tests, and end-to-end tests. When you run tests against cloud-based code that also interacts with cloud-based services, you get the most accurate measure of quality of your code.

A convenient way to run a Lambda function in the cloud is with a test event in the AWS Management Console. A test event is a JSON input to your function. If your function does not require input, the event can be an empty JSON document ( `{}` ). The console provides sample
events for a variety of service integrations. After creating an event in the console, you can also share it with your team to make testing easier and consistent.

Learn how to [debug a sample function in the console](#).

**Note**

Although running functions in the console is a quick way to debug, **automating** your test cycles is essential to increase application quality and development speed.

Test automation samples are available in the [Serverless Test Samples repository](#). The following command line runs an automated **Python integration test example**:

```
python -m pytest -s tests/integration -v
```

Although the test runs locally, it interacts with cloud-based resources. These resources have been deployed using the AWS Serverless Application Model and AWS SAM command line tool. The test code first retrieves the deployed stack outputs, which includes the API endpoint, function ARN, and security role. Next, the test sends a request to the API endpoint, which responds with a list of Amazon S3 buckets. This test runs entirely against cloud-based resources to verify those resources are deployed, secured, and work as expected.

```
========================= test session starts =========================
platform darwin -- Python 3.10.10, pytest-7.3.1, pluggy-1.0.0
-- /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-lambda/venv/bin/python
cachedir: .pytest_cache
rootdir: /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-lambda
plugins: mock-3.10.0
collected 1 item

tests/integration/test_api_gateway.py::TestApiGateway::test_api_gateway

--> Stack outputs:

HelloWorldApi
  = https://p7teqs3162.execute-api.us-west-2.amazonaws.com/Prod/hello/
> API Gateway endpoint URL for Prod stage for Hello World function
```
For cloud-native application development, testing in the cloud provides the following benefits:

- You can test **every** available service.
- You are always using the most recent service APIs and return values.
- A cloud test environment closely resembles your production environment.
- Tests can cover security policies, service quotas, configurations and infrastructure specific parameters.
- Every developer can quickly create one or more testing environments in the cloud.
- Cloud tests increase confidence your code will run correctly in production.

Testing in the cloud does have some disadvantages. The most obvious negative of testing in the cloud is that deployments to cloud environments typically take longer than deployments to a local desktop environments.

Fortunately, tools such as **AWS Serverless Application Model (AWS SAM) Accelerate**, **AWS Cloud Development Kit (AWS CDK) watch mode**, and **SST** (3rd party) reduce the latency involved with cloud deployment iterations. These tools can monitor your infrastructure and code and automatically deploy incremental updates into your cloud environment.
Unlike local testing, testing in the cloud requires additional resources which may incur service costs. Creating isolated testing environments may increase the burden on your DevOps teams, especially in organizations with strict controls around accounts and infrastructure. Even so, when working with complex infrastructure scenarios, the cost in developer time to set up and maintain an intricate local environment could be similar (or more costly) than using disposable testing environments created with Infrastructure as Code automation tools.

Testing in the cloud, even with these considerations, is still the best way to guarantee the quality of your serverless solutions.

**Testing with mocks**

Testing with mocks is a technique where you create replacement objects in your code to simulate the behavior of a cloud service.

For example, you could write a test that uses a mock of the Amazon S3 service that returns a specific response whenever the `CreateObject` method is called. When a test runs, the mock returns that programmed response without calling Amazon S3, or any other service endpoints.

Mock objects are often generated by a mock framework to reduce development effort. Some mock frameworks are generic and others are designed specifically for AWS SDKs, such as Moto, a Python library for mocking AWS services and resources.

Note that mock objects differ from emulators in that mocks are typically created or configured by a developer as part of the test code, whereas emulators are standalone applications that expose functionality in the same manner as the systems they emulate.

The advantages of using mocks include the following:

- Mocks can simulate third-party services that are beyond the control of your application, such as APIs and software as a service (SaaS) providers, without needing direct access to those services.
- Mocks are useful for testing failure conditions, especially when such conditions are hard to simulate, like a service outage.
• Mock can provide fast local testing once configured.
• Mocks can provide substitute behavior for virtually any kind of object, so mocking strategies can create coverage for a wider variety of services than emulators.
• When new features or behaviors become available, mock testing can react more quickly. By using a generic mock framework, you can simulate new features as soon as the updated AWS SDK become available.

Mock testing has these disadvantages:

• Mocks generally require a non-trivial amount of setup and configuration effort, specifically when trying to determine return values from different services in order to properly mock responses.
• Mocks are written, configured, and must be maintained by developers, increasing their responsibilities.
• You might need to have access to the cloud in order to understand the APIs and return values of services.
• Mocks can be difficult to maintain. When mocked cloud API signatures change, or return value schemas evolve, you need to update your mocks. Mocks also require updates if you extend your application logic to make calls to new APIs.
• Tests that use mocks might pass in desktop environments but fail in the cloud. Results may not match the current API. Service configuration and quotas cannot be tested.
• Mock frameworks are limited in testing or detecting AWS Identity and Access Management (IAM) policy or quota limitations. Although mocks are better at simulating when authorization fails or a quota is exceeded, testing cannot determine which outcome will actually occur in a production environment.

Testing with emulation

Emulators are typically a locally running application which mimics a production AWS service.

Emulators have APIs that are similar to their cloud counterparts and provide similar return values. They can also simulate state changes that are initiated by API calls. For example, you might use AWS SAM to run a function with AWS SAM local to emulate the Lambda service so that you can quickly invoke a function. See AWS SAM local in the AWS Serverless Application Model Developer Guide for details.

The advantages of test with emulators include the following:

• Emulators can facilitate fast local development iterations and testing.
Emulators provide a familiar environment for developers used to developing code in a local environment. For example, if you’re familiar with the development of an n-tier application, you might have a database engine and web server, similar to those running in production, running on your local machine to provide quick, local, isolated test capability.

Emulators do not require any changes to cloud infrastructure (such as developer cloud accounts), so it’s easy to implement with existing testing patterns.

Testing with emulators has these disadvantages:

- Emulators can be difficult to set up and replicate, especially when used in CI/CD pipelines. This can increase the workload of IT staff or developers who manage their own software.
- Emulated features and APIs typically lag behind service updates. This can lead to errors because tested code does not match the actual API, and impede the adoption of new features.
- Emulators require support, updates, bug fixes, and feature parity enhancements. These are the responsibility of the emulator author, which could be a third-party company.
- Tests that rely on emulators may provide successful results locally, but fail in the cloud due to production security policies, inter-service configurations, or exceeding Lambda quotas.
- Many AWS services do not have emulators available. If you rely on emulation, you may not have a satisfactory testing option for portions of your application.

**Best practices**

The following sections provide recommendations for successful serverless application testing.

You can find practical examples of tests and test automation in the [Serverless Test Samples repository](#).

**Prioritize testing in the cloud**

Testing in the cloud provides the most reliable, accurate, and complete test coverage. Performing tests in the context of the cloud will comprehensively test not only business logic but also security policies, service configurations, quotas, and the most up to date API signatures and return values.

**Structure your code for testability**

Simplify your tests and Lambda functions by separating Lambda-specific code from your core business logic.
Your Lambda function handler should be a slim adapter that takes in event data and passes only the details that matter to your business logic method(s). With this strategy, you can wrap comprehensive tests around your business logic without worrying about Lambda-specific details. Your AWS Lambda functions should not require setting up a complex environment or large amount of dependencies to create and initialize the component under test.

Generally speaking, you should write a handler that extracts and validates data from the incoming event and context objects, then sends that input to methods that perform your business logic.

**Accelerate development feedback loops**

There are tools and techniques to accelerate development feedback loops. For example, AWS SAM Accelerate and AWS CDK watch mode both decrease the time required to update cloud environments.

The samples in the GitHub Serverless Test Samples repository explore some of these techniques.

We also recommend that you create and test cloud resources as early as possible during development—not only after a check-in to source control. This practice enables quicker exploration and experimentation when developing solutions. In addition, automating deployment from a development machine helps you discover cloud configuration problems more quickly and reduces wasted effort for updates and code review processes.

**Focus on integration tests**

When building applications with Lambda, testing components together is a best practice.

Tests that run against two or more architectural components are called integration tests. The goal of integration tests is to understand not only how your code will execute across components, but how the environment hosting your code will behave. End-to-end tests are special types of integration tests that verify behaviors across an entire application.

To build integration tests, deploy your application to a cloud environment. This can be done from a local environment or through a CI/CD pipeline. Then, write tests to exercise the system under test (SUT) and validate expected behavior.

For example, the system under test could be an application that uses API Gateway, Lambda and DynamoDB. A test could make a synthetic HTTP call to an API Gateway endpoint and validate that the response included the expected payload. This test validates that the AWS Lambda code
is correct, and that each service is correctly configured to handle the request, including the IAM permissions between them. Further, you could design the test to write records of various sizes to verify your service quotas, such as max record size in DynamoDB, are set up correctly.

Create isolated test environments

Testing in the cloud typically requires isolated developer environments, so that tests, data, and events do not overlap.

One approach is to provide each developer a dedicated AWS account. This will avoid conflicts with resource naming that can occur when multiple developers working in a shared code base, attempt to deploy resources or invoke an API.

Automated test processes should create uniquely named resources for each stack. For example, you can set up scripts or TOML configuration files so that AWS SAM CLI `sam deploy` or `sam sync` commands will automatically specify a stack with a unique prefix.

In some cases, developers share an AWS account. This may be due to having resources in your stack that are expensive to operate, or to provision and configure. For example, a database may be shared to make it easier to set up and seed the data properly.

If developers share an account, you should set boundaries to identify ownership and eliminate overlap. One way to do this is by prefixing stack names with developer user IDs. Another popular approach is to set up stacks based on code branches. With branch boundaries, environments are isolated, but developers can still share resources, such as a relational database. This approach is a best practice when developers work on more than one branch at a time.

Testing in the cloud is valuable for all phases of testing, including unit tests, integration tests, and end-to-end tests. Maintaining proper isolation is essential; but you still want your QA environment to resemble your production environment as closely as possible. For this reason, teams add change control processes for QA environments.
For pre-production and production environments, boundaries are typically drawn at the account level to insulate workloads from noisy neighbor problems and implement least privilege security controls to protect sensitive data. Workloads have quotas. You don’t want your testing to consume quotas allocated for production (noisy neighbor) or have access to customer data. Load testing is another activity you should isolate from your production stack.

In all cases, environments should be configured with alerts and controls to avoid unnecessary spending. For example, you can limit the type, tier, or size of resources that can be created, and set up email alerts when estimated costs exceed a given threshold.

**Use mocks for isolated business logic**

Mock frameworks are a valuable tool for writing fast unit tests. They are especially beneficial when tests cover complex internal business logic, such as mathematical or financial calculations or simulations. Look for unit tests that have a large number of test cases or input variations, where those inputs do not change the pattern or the content of calls to other cloud services.

Code that is covered by unit tests with mocks should also be covered by testing in the cloud. This is recommended because a developer laptop or build machine environment could be configured differently than a production environment in the cloud. For example, your Lambda functions could use more memory or time than allocated when run with certain input parameters. Or your code might include environment variables that are not configured in the same way (or at all), and the differences could cause the code to behave differently or fail.

The benefit of mocks is less for integration tests, because the level of effort to implement the necessary mocks increases with the number of connection points. End-to-end testing should not use mocks, because these tests generally deal with states and complex logic that cannot be easily simulated with mock frameworks.

Lastly, avoid using mocked cloud services to validate the proper implementation of service calls. Instead, make cloud service calls in the cloud to validate behavior, configuration, and functional implementation.

**Use emulators sparingly**

Emulators can be convenient for some use cases, for example, for a development team with limited, unreliable, or slow internet access. But, in most circumstances, choose to use emulators sparingly.
By avoiding emulators, you will be able to build and innovate with the latest service features and up to date APIs. You will not be stuck waiting on vendor releases to achieve feature parity. You will reduce your upfront and ongoing expenses for purchasing and configuration on multiple development systems and build machines. Moreover, you will avoid the problem that many cloud services simply do not have emulators available. A testing strategy that depends on emulation will make it impossible to use those services (leading to potentially more expensive workarounds) or produce code and configurations that aren't well tested.

When you do use emulation for testing, you must still test in the cloud to verify configuration and to test interactions with cloud services that can only be simulated or mocked in an emulated environment.

**Challenges testing locally**

When you use emulators and mocked calls to test on your local desktop you might experience testing inconsistencies as your code progresses from environment to environment in your CI/CD pipeline. Unit tests to validate your application's business logic on your desktop may not accurately test critical aspects of the cloud services.

The following examples provide cases to watch out for when testing locally with mocks and emulators:

**Example: Lambda function creates an S3 bucket**

If a Lambda function's logic depends on creating an S3 bucket, a complete test should confirm that Amazon S3 was called and the bucket was successfully created.

- In a mock testing setup, you might mock a success response and potentially add a test case to handle a failure response.
- In an emulation testing scenario, the `CreateBucket` API might be called, but you need to be aware that the identity making the local call will **not** originate from the Lambda service. The calling identity will not assume a security role as it would in the cloud, so a placeholder authentication will be used instead, possibly with a more permissive role or user identity that will be different when run in the cloud.

The mock and emulation setups will test what the Lambda function will do if it calls Amazon S3; however, those tests will not verify that the Lambda function, as configured, is capable of successfully creating the Amazon S3 bucket. You must make sure the role assigned to the function
has an attached security policy that allows the function to perform the `s3:CreateBucket` action. If not, the function will likely fail when deployed to a cloud environment.

**Example: Lambda function processes messages from an Amazon SQS queue**

If an Amazon SQS queue is the source of a Lambda function, a complete test should verify that the Lambda function is successfully invoked when a message is put in a queue.

Emulation testing and mock testing are generally set up to run the Lambda function code directly, and to simulate the Amazon SQS integration by passing a JSON event payload (or a deserialized object) as the function handler's input.

Local testing that simulates the Amazon SQS integration will test what the Lambda function will do when it's called by Amazon SQS with a given payload, but the test will not verify that Amazon SQS will successfully invoke the Lambda function when it is deployed to a cloud environment.

Some examples of configuration problems you might encounter with Amazon SQS and Lambda include the following:

- Amazon SQS visibility timeout is too low, resulting in multiple invocations when only one was intended.
- The Lambda function's execution role doesn't allow reading messages from the queue (through `sqs:ReceiveMessage`, `sqs:DeleteMessage`, or `sqs:GetQueueAttributes`).
- The sample event that is passed to the Lambda function exceeds the Amazon SQS message size quota. Therefore, the test is invalid because Amazon SQS would never be able to send a message of that size.

As these examples show, tests that cover business logic but not the configurations between cloud services are likely to provide unreliable results.

**FAQ**

**I have a Lambda function that performs calculations and returns a result without calling any other services. Do I really need to test it in the cloud?**

Yes. Lambda functions have configuration parameters that could change the outcome of the test. All Lambda function code has a dependency on **timeout and memory settings**, which could cause the function to fail if those settings are not set properly. Lambda policies also enable standard...
output logging to Amazon CloudWatch. Even if your code does not call CloudWatch directly, permission is needed to enable logging. This required permission cannot be accurately mocked or emulated.

**How can testing in the cloud help with unit testing? If it’s in the cloud and connects to other resources, isn’t that an integration test?**

We define *unit tests* as tests that operate on architectural components in isolation, but this does not prevent tests from including components that may call other services or use some network communication.

Many serverless applications have architectural components that can be tested in isolation, even in the cloud. One example is a Lambda function that takes input, processes the data, and sends a message to an Amazon SQS queue. A unit test of this function would likely test whether input values result in certain values being present in the queued message.

Consider a test that is written by using the Arrange, Act, Assert pattern:

- **Arrange**: Allocate resources (a queue to receive messages, and the function under test).
- **Act**: Call the function under test.
- **Assert**: Retrieve the message sent by the function, and validate the output.

A mock testing approach would involve mocking the queue with an in-process mock object, and creating an in-process instance of the class or module that contains the Lambda function code. During the Assert phase, the queued message would be retrieved from the mocked object.

In a cloud-based approach, the test would create an Amazon SQS queue for the purposes of the test, and would deploy the Lambda function with environment variables that are configured to use the isolated Amazon SQS queue as the output destination. After running the Lambda function, the test would retrieve the message from the Amazon SQS queue.

The cloud-based test would run the same code, assert the same behavior, and validate the application’s functional correctness. However, it would have the added advantage of being able to validate the settings of the Lambda function: the IAM role, IAM policies, and the function’s timeout and memory settings.

**Next steps and resources**

Use the following resources to learn more and explore practical examples of testing.
Sample implementations

The [Serverless Test Samples repository](https://github.com/aws-labs/serverless-test-samples) on GitHub contains concrete examples of tests that follow the patterns and best practices described in this guide. The repository contains sample code and guided walkthroughs of the mock, emulation, and cloud testing processes described in previous sections. Use this repository to get up to speed on the latest serverless testing guidance from AWS.

Further reading

Visit [Serverless Land](https://www.serverlessland.com) to access the latest blogs, videos, and training for AWS serverless technologies.

The following AWS blog posts are also recommended reading:

- [Increasing development speed with CDK Watch](https://aws.amazon.com/blogs/serverless/increasing-development-speed-with-cdk-watch/) (AWS blog post)
- [Getting started with testing serverless applications](https://aws.amazon.com/blogs/serverless/getting-started-with-testing-serverless-applications/) (AWS blog post)

Tools

- AWS SAM – [Integrating with automated tests](https://aws.amazon.com/blogs/serverless/integrating-with-automated-tests/)
Building Lambda functions with Node.js

You can run JavaScript code with Node.js in AWS Lambda. Lambda provides runtimes for Node.js that run your code to process events. Your code runs in an environment that includes the AWS SDK for JavaScript, with credentials from an AWS Identity and Access Management (IAM) role that you manage.

Lambda supports the following Node.js runtimes.

Node.js

<table>
<thead>
<tr>
<th>Name</th>
<th>Identifier</th>
<th>SDK</th>
<th>Operating system</th>
<th>Deprecation date</th>
<th>Block function create</th>
<th>Block function update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node.js 20</td>
<td>nodejs20.x</td>
<td>3.362.0</td>
<td>Amazon Linux</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Node.js 18</td>
<td>nodejs18.x</td>
<td>3.362.0</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Node.js 16</td>
<td>nodejs16.x</td>
<td>2.1374.0</td>
<td>Amazon Linux 2</td>
<td>Jun 12, 2024</td>
<td>Jul 15, 2024</td>
<td>Aug 15, 2024</td>
</tr>
</tbody>
</table>

Note

Node.js 18 and later runtimes use AWS SDK for JavaScript v3. To migrate a function from an earlier runtime, follow the migration workshop on GitHub. For more information about AWS SDK for JavaScript version 3, see the Modular AWS SDK for JavaScript is now generally available blog post.

To create a Node.js function

1. Open the Lambda console.
2. Choose Create function.
3. Configure the following settings:

- **Function name**: Enter a name for the function.
- **Runtime**: Choose Node.js 20.x.

4. Choose **Create function**.

5. To configure a test event, choose **Test**.

6. For **Event name**, enter **test**.

7. Choose **Save changes**.

8. To invoke the function, choose **Test**.

The console creates a Lambda function with a single source file named `index.js` or `index.mjs`. You can edit this file and add more files in the built-in code editor. To save your changes, choose **Save**. Then, to run your code, choose **Test**.

Note

The Lambda console uses AWS Cloud9 to provide an integrated development environment in the browser. You can also use AWS Cloud9 to develop Lambda functions in your own environment. For more information, see [Working with AWS Lambda functions using the AWS Toolkit](https://docs.aws.amazon.com/lambda/latest/dg/working-with-lambda.html) in the AWS Cloud9 user guide.

The `index.js` or `index.mjs` file exports a function named `handler` that takes an event object and a context object. This is the **handler function** that Lambda calls when the function is invoked. The Node.js function runtime gets invocation events from Lambda and passes them to the handler. In the function configuration, the handler value is `index.handler`.

When you save your function code, the Lambda console creates a .zip file archive deployment package. When you develop your function code outside of the console (using an IDE) you need to create a deployment package to upload your code to the Lambda function.

Note

To get started with application development in your local environment, deploy one of the sample applications available in this guide's GitHub repository.
Sample Lambda applications in Node.js

- **blank-nodejs** – A Node.js function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.
- **nodejs-apig** – A function with a public API endpoint that processes an event from API Gateway and returns an HTTP response.
- **rds-mysql** – A function that relays queries to a MySQL for RDS Database. This sample includes a private VPC and database instance configured with a password in AWS Secrets Manager.
- **efs-nodejs** – A function that uses an Amazon EFS file system in a Amazon VPC. This sample includes a VPC, file system, mount targets, and access point configured for use with Lambda.
- **list-manager** – A function processes events from an Amazon Kinesis data stream and update aggregate lists in Amazon DynamoDB. The function stores a record of each event in a MySQL for RDS Database in a private VPC. This sample includes a private VPC with a VPC endpoint for DynamoDB and a database instance.
- **error-processor** – A Node.js function generates errors for a specified percentage of requests. A CloudWatch Logs subscription invokes a second function when an error is recorded. The processor function uses the AWS SDK to gather details about the request and stores them in an Amazon S3 bucket.

The function runtime passes a context object to the handler, in addition to the invocation event. The **context object** contains additional information about the invocation, the function, and the execution environment. More information is available from environment variables.

Your Lambda function comes with a CloudWatch Logs log group. The function runtime sends details about each invocation to CloudWatch Logs. It relays any **logs that your function outputs** during invocation. If your function **returns an error**, Lambda formats the error and returns it to the invoker.

**Node.js initialization**

Node.js has a unique event loop model that causes its initialization behavior to be different from other runtimes. Specifically, Node.js uses a non-blocking I/O model that supports asynchronous operations. This model allows Node.js to perform efficiently for most workloads. For example,
if a Node.js function makes a network call, that request may be designated as an asynchronous operation and placed into a callback queue. The function may continue to process other operations within the main call stack without getting blocked by waiting for the network call to return. Once the network call is completed, its callback is executed and then removed from the callback queue.

Some initialization tasks may run asynchronously. These asynchronous tasks are not guaranteed to complete execution prior to an invocation. For example, code that makes a network call to fetch a parameter from AWS Parameter Store may not be complete by the time Lambda executes the handler function. As a result, the variable may be null during an invocation. To avoid this, ensure that variables and other asynchronous code are fully initialized before continuing with the rest of the function's core business logic.

Alternatively, you can designate your function code as an ES module, allowing you to use `await` at the top level of the file, outside the scope of your function handler. When you `await` every `Promise`, the asynchronous initialization code completes before handler invocations, maximizing the effectiveness of provisioned concurrency in reducing cold start latency. For more information and an example, see Using Node.js ES modules and top-level await in AWS Lambda.

**Designating a function handler as an ES module**

By default, Lambda treats files with the `.js` suffix as CommonJS modules. Optionally, you can designate your code as an ES module. You can do this in two ways: specifying the `type` as `module` in the function's `package.json` file, or by using the `.mjs` file name extension. In the first approach, your function code treats all `.js` files as ES modules, while in the second scenario, only the file you specify with `.mjs` is an ES module. You can mix ES modules and CommonJS modules by naming them `.mjs` and `.cjs` respectively, as `.mjs` files are always ES modules and `.cjs` files are always CommonJS modules.

For Node.js runtime versions up to Node.js 16, the Lambda runtime loads ES modules from the same folder as your function handler, or a subfolder. Starting with Node.js 18, Lambda searches folders in the `NODE_PATH` environment variable when loading ES modules. Also, starting with Node.js 18, you can load the AWS SDK that's included in the runtime using ES module `import` statements. You can also load ES modules from layers.

**Using keep-alive for TCP connections**

The default Node.js HTTP/HTTPS agent creates a new TCP connection for every new request. To avoid the cost of establishing new connections, you can use `keepAlive: true` to reuse
connections that your function makes using the AWS SDK for JavaScript. Keep-alive can reduce request times for Lambda functions that make multiple API calls using the SDK. Keep-alive behavior differs depending on which version of the SDK you're using:

- In the AWS SDK for JavaScript 3.x, which is included in nodejs18.x and later Lambda runtimes, keep-alive is enabled by default. To disable keep-alive, see Reusing connections with keep-alive in Node.js in the AWS SDK for JavaScript 3.x Developer Guide.

- In the AWS SDK for JavaScript 2.x, which is included in the nodejs16.x Lambda runtime, keep-alive is disabled by default. To enable keep-alive, see Reusing Connections with Keep-Alive in Node.js in the AWS SDK for JavaScript 2.x Developer Guide.

For more information about using keep-alive, see HTTP keep-alive is on by default in modular AWS SDK for JavaScript on the AWS Developer Tools Blog.

**CA certificate loading**

For Node.js runtime versions up to Node.js 18, Lambda automatically loads Amazon-specific CA (certificate authority) certificates to make it easier for you to create functions that interact with other AWS services. For example, Lambda includes the Amazon RDS certificates necessary for validating the server identity certificate installed on your Amazon RDS database. This behavior can have a performance impact during cold starts.

Starting with Node.js 20, Lambda no longer loads additional CA certificates by default. The Node.js 20 runtime contains a certificate file with all Amazon CA certificates located at /var/runtime/ca-cert.pem. To restore the same behavior from Node.js 18 and earlier runtimes, set the NODE_EXTRA_CA_CERTS environment variable to /var/runtime/ca-cert.pem.

For optimal performance, we recommend bundling only the certificates that you need with your deployment package and loading them via the NODE_EXTRA_CA_CERTS environment variable. The certificates file should consist of one or more trusted root or intermediate CA certificates in PEM format. For example, for RDS, include the required certificates alongside your code as certificates/rds.pem. Then, load the certificates by setting NODE_EXTRA_CA_CERTS to /var/task/certificates/rds.pem.

**Topics**

- AWS Lambda function handler in Node.js
- Deploy Node.js Lambda functions with .zip file archives
• **Deploy Node.js Lambda functions with container images**
• **AWS Lambda context object in Node.js**
• **AWS Lambda function logging in Node.js**
• **AWS Lambda function errors in Node.js**
• **Instrumenting Node.js code in AWS Lambda**
AWS Lambda function handler in Node.js

The Lambda function handler is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

The following example function logs the contents of the event object and returns the location of the logs.

### Note

This page shows examples of both CommonJS and ES module handlers. To learn about the difference between these two handler types, see [Designating a function handler as an ES module](#).

#### CommonJS module handler

**Example**

```javascript
exports.handler = async function (event, context) {
    console.log("EVENT: \n" + JSON.stringify(event, null, 2));
    return context.logStreamName;
};
```

#### ES module handler

**Example**

```javascript
export const handler = async (event, context) => {
    console.log("EVENT: \n" + JSON.stringify(event, null, 2));
    return context.logStreamName;
};
```

When you configure a function, the value of the handler setting is the file name and the name of the exported handler method, separated by a dot. The default in the console and for examples in this guide is `index.handler`. This indicates the `handler` method that's exported from the `index.js` file.
The runtime passes arguments to the handler method. The first argument is the event object, which contains information from the invoker. The invoker passes this information as a JSON-formatted string when it calls `Invoke`, and the runtime converts it to an object. When an AWS service invokes your function, the event structure varies by service.

The second argument is the context object, which contains information about the invocation, function, and execution environment. In the preceding example, the function gets the name of the log stream from the context object and returns it to the invoker.

You can also use a callback argument, which is a function that you can call in non-async handlers to send a response. We recommend that you use async/await instead of callbacks. Async/await provides improved readability, error handling, and efficiency. For more information about the differences between async/await and callbacks, see Using callbacks.

**Using async/await**

If your code performs an asynchronous task, use the async/await pattern to make sure that the handler finishes running. Async/await is a concise and readable way to write asynchronous code in Node.js, without the need for nested callbacks or chaining promises. With async/await, you can write code that reads like synchronous code, while still being asynchronous and non-blocking.

The async keyword marks a function as asynchronous, and the await keyword pauses the execution of the function until a Promise is resolved.

**Note**

Make sure to wait for asynchronous events to complete. If the function returns before async events are complete, the function might fail or cause unexpected behavior in your application. This can happen when a `forEach` loop contains an async event. `forEach` loops expect a synchronous call. For more information, see `Array.prototype.forEach()` in the Mozilla documentation.

**CommonJS module handler**

**Example – HTTP request with async/await**

```javascript
const https = require("https");
let url = "https://aws.amazon.com/";
```
exports.handler = async function (event) {
    let statusCode;
    await new Promise(function (resolve, reject) {
        https.get(url, (res) => {
            statusCode = res.statusCode;
            resolve(statusCode);
        }).on("error", (e) => {
            reject(Error(e));
        });
    });
    console.log(statusCode);
    return statusCode;
};

ES module handler

Example – HTTP request with async/await

This example uses fetch, which is available in the nodejs18.x runtime.

const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        // fetch is available with Node.js 18
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    } catch (e) {
        console.error(e);
        return 500;
    }
};

The next example uses async/await to list your Amazon Simple Storage Service buckets.

Note
Before using this example, make sure that your function's execution role has Amazon S3 read permissions.
CommonJS module handler

**Example – AWS SDK v2 with async/await**

This example uses the [AWS SDK for JavaScript v2](https://aws.amazon.com/sdk-for-javascript/), which is included in the nodejs16.x Lambda runtime.

```javascript
const AWS = require('aws-sdk')
const s3 = new AWS.S3()

exports.handler = async function(event) {
    const buckets = await s3.listBuckets().promise()
    return buckets
}
```

ES module handler

**Example – AWS SDK v3 with async/await**

This example uses the [AWS SDK for JavaScript v3](https://aws.amazon.com/sdk-for-javascript/), which is available in the nodejs18.x runtime.

```javascript
import {S3Client, ListBucketsCommand} from '@aws-sdk/client-s3';
const s3 = new S3Client({region: 'us-east-1'});

export const handler = async(event) => {
    const data = await s3.send(new ListBucketsCommand({}));
    return data.Buckets;
};
```

**Using callbacks**

We recommend that you use `async/await` to declare the function handler instead of using callbacks. Async/await is a better choice for several reasons:

- **Readability:** Async/await code is easier to read and understand than callback code, which can quickly become difficult to follow and result in callback hell.
- **Debugging and error handling:** Debugging callback-based code can be difficult. The call stack can become hard to follow and errors can easily be swallowed. With async/await, you can use try/catch blocks to handle errors.
• **Efficiency**: Callbacks often require switching between different parts of the code. Async/await can reduce the number of context switches, resulting in more efficient code.

When you use callbacks in your handler, the function continues to execute until the `event loop` is empty or the function times out. The response isn't sent to the invoker until all event loop tasks are finished. If the function times out, an error is returned instead. You can configure the runtime to send the response immediately by setting `context.callbackWaitsForEmptyEventLoop` to false.

The callback function takes two arguments: an `Error` and a response. The response object must be compatible with `JSON.stringify`.

The following example function checks a URL and returns the status code to the invoker.

**CommonJS module handler**

**Example – HTTP request with callback**

```javascript
const https = require("https");
let url = "https://aws.amazon.com/";

exports.handler = function (event, context, callback) {
  https.get(url, (res) => {
    callback(null, res.statusCode);
  }).on("error", (e) => {
    callback(Error(e));
  });
};
```

**ES module handler**

**Example – HTTP request with callback**

```javascript
import https from "https";
let url = "https://aws.amazon.com/";

export function handler(event, context, callback) {
  https.get(url, (res) => {
    callback(null, res.statusCode);
  }).on("error", (e) => {
    callback(Error(e));
  });
};
```
In the next example, the response from Amazon S3 is returned to the invoker as soon as it's available. The timeout running on the event loop is frozen, and it continues running the next time the function is invoked.

⚠️ Note
Before using this example, make sure that your function's execution role has Amazon S3 read permissions.

CommonJS module handler

**Example – AWS SDK v2 with callbackWaitsForEmptyEventLoop**

This example uses the [AWS SDK for JavaScript v2](https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/), which is included in the nodejs16.x Lambda runtime.

```javascript
const AWS = require("aws-sdk");
const s3 = new AWS.S3();

exports.handler = function (event, context, callback) {
  context.callbackWaitsForEmptyEventLoop = false;
  s3.listBuckets(null, callback);
  setTimeout(function () {
    console.log("Timeout complete.");
  }, 5000);
};
```

ES module handler

**Example – AWS SDK v3 with callbackWaitsForEmptyEventLoop**

This example uses the [AWS SDK for JavaScript v3](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/), which is available in the nodejs18.x runtime.

```javascript
import AWS from "@aws-sdk/client-s3";
const s3 = new AWS.S3({});
```
export const handler = function (event, context, callback) {
  context.callbackWaitsForEmptyEventLoop = false;
  s3.listBuckets({}, callback);
  setTimeout(function () {
    console.log("Timeout complete.");
  }, 5000);
};
Deploy Node.js Lambda functions with .zip file archives

Your AWS Lambda function's code comprises a .js or .mjs file containing your function's handler code, together with any additional packages and modules your code depends on. To deploy this function code to Lambda, you use a deployment package. This package may either be a .zip file archive or a container image. For more information about using container images with Node.js, see Deploy Node.js Lambda functions with container images.

To create your deployment package as .zip file archive, you can use your command-line tool's built-in .zip file archive utility, or any other .zip file utility such as 7zip. The examples shown in the following sections assume you're using a command-line zip tool in a Linux or MacOS environment. To use the same commands in Windows, you can install the Windows Subsystem for Linux to get a Windows-integrated version of Ubuntu and Bash.

Note that Lambda uses POSIX file permissions, so you may need to set permissions for the deployment package folder before you create the .zip file archive.

Topics

- Runtime dependencies in Node.js
- Creating a .zip deployment package with no dependencies
- Creating a .zip deployment package with dependencies
- Creating a Node.js layer for your dependencies
- Dependency search path and runtime-included libraries
- Creating and updating Node.js Lambda functions using .zip files

Runtime dependencies in Node.js

For Lambda functions that use the Node.js runtime, a dependency can be any Node.js module. The Node.js runtime includes a number of common libraries, as well as a version of the AWS SDK for JavaScript. The nodejs16.x Lambda runtime includes version 2.x of the SDK. Runtime versions nodejs18.x and later include version 3 of the SDK. To use version 2 of the SDK with runtime versions nodejs18.x and later, add the SDK to your .zip file deployment package. If your chosen runtime includes the version of the SDK you are using, you don't need to include the SDK library in your .zip file.

Lambda periodically updates the SDK libraries in the Node.js runtime to include the latest features and security upgrades. Lambda also applies security patches and updates to the other libraries.
included in the runtime. To have full control of the dependencies in your package, you can add your preferred version of any runtime-included dependency to your deployment package. For example, if you want to use a particular version of the SDK for JavaScript, you can include it in your .zip file as a dependency. For more information on adding runtime-included dependencies to your .zip file, see Dependency search path and runtime-included libraries.

Under the AWS shared responsibility model, you are responsible for the management of any dependencies in your functions' deployment packages. This includes applying updates and security patches. To update dependencies in your function's deployment package, first create a new .zip file and then upload it to Lambda. See Creating a .zip deployment package with dependencies and Creating and updating Node.js Lambda functions using .zip files for more information.

Creating a .zip deployment package with no dependencies

If your function code has no dependencies except for libraries included in the Lambda runtime, your .zip file contains only the index.js or index.mjs file with your function's handler code. Use your preferred zip utility to create a .zip file with your index.js or index.mjs file at the root. If the file containing your handler code isn't at the root of your .zip file, Lambda won't be able to run your code.

To learn how to deploy your .zip file to create a new Lambda function or update an existing one, see Creating and updating Node.js Lambda functions using .zip files.

Creating a .zip deployment package with dependencies

If your function code depends on packages or modules that aren't included in the Lambda Node.js runtime, you can either add these dependencies to your .zip file with your function code or use a Lambda layer. The instructions in this section show you how to include your dependencies in your .zip deployment package. For instructions on how to include your dependencies in a layer, see the section called “Creating a Node.js layer for your dependencies”.

The following example CLI commands create a .zip file named my_deployment_package.zip containing the index.js or index.mjs file with your function's handler code and its dependencies. In the example, you install dependencies using the npm package manager.

To create the deployment package

1. Navigate to the project directory containing your index.js or index.mjs source code file. In this example, the directory is named my_function.
cd my_function

2. Install your function’s required libraries in the node_modules directory using the npm install command. In this example you install the AWS X-Ray SDK for Node.js.

```bash
npm install aws-xray-sdk
```

This creates a folder structure similar to the following:

```
~/my_function
### index.mjs
### node_modules
    ### async
    ### async-listener
    ### atomic-batcher
    ### aws-sdk
    ### aws-xray-sdk
    ### aws-xray-sdk-core
```

You can also add custom modules that you create yourself to your deployment package. Create a directory under node_modules with the name of your module and save your custom written packages there.

3. Create a .zip file that contains the contents of your project folder at the root. Use the `r` (recursive) option to ensure that zip compresses the subfolders.

```bash
zip -r my_deployment_package.zip .
```

## Creating a Node.js layer for your dependencies

The instructions in this section show you how to include your dependencies in a layer. For instructions on how to include your dependencies in your deployment package, see the section called “Creating a .zip deployment package with dependencies”.

When you add a layer to a function, Lambda loads the layer content into the /opt directory of that execution environment. For each Lambda runtime, the PATH variable already includes specific folder paths within the /opt directory. To ensure that your layer content gets picked up by the PATH variable, include the content in the following folder paths:
• nodejs/node_modules
• nodejs/node16/node_modules (NODE_PATH)
• nodejs/node18/node_modules (NODE_PATH)
• nodejs/node20/node_modules (NODE_PATH)

For example, your layer .zip file structure might look like the following:

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

In addition, Lambda automatically detects any libraries in the /opt/lib directory, and any binaries in the /opt/bin directory. To ensure that Lambda properly finds your layer content, you can also create a layer with the following structure:

```
custom-layer.zip
# lib
 | lib_1
 | lib_2
# bin
 | bin_1
 | bin_2
```

After you package your layer, see the section called “Creating and deleting layers” and the section called “Adding layers” to complete your layer setup.

**Dependency search path and runtime-included libraries**

The Node.js runtime includes a number of common libraries, as well as a version of the AWS SDK for JavaScript. If you want to use a different version of a runtime-included library, you can do this by bundling it with your function or by adding it as a dependency in your deployment package. For example, you can use a different version of the SDK by adding it to your .zip deployment package. You can also include it in a Lambda layer for your function.

When you use an import or require statement in your code, the Node.js runtime searches the directories in the NODE_PATH path until it finds the module. By default, the first location the runtime searches is the directory into which your .zip deployment package is decompressed and mounted (/var/task). If you include a version of a runtime-included library in your deployment
package, this version will take precedence over the version included in the runtime. Dependencies in your deployment package also have precedence over dependencies in layers.

When you add a dependency to a layer, Lambda extracts this to /opt/nodejs/nodexx/node_modules where nodexx represents the version of the runtime you are using. In the search path, this directory has precedence over the directory containing the runtime-included libraries (/var/lang/lib/node_modules). Libraries in function layers therefore have precedence over versions included in the runtime.

You can see the full search path for your Lambda function by adding the following line of code.

```javascript
console.log(process.env.NODE_PATH)
```

You can also add dependencies in a separate folder inside your .zip package. For example, you might add a custom module to a folder in your .zip package called common. When your .zip package is decompressed and mounted, this folder is placed inside the /var/task directory. To use a dependency from a folder in your .zip deployment package in your code, use an `import { } from` or `const { } = require()` statement, depending on whether you are using CJS or ESM module resolution. For example:

```javascript
import { myModule } from './common'
```

If you bundle your code with esbuild, rollup, or similar, the dependencies used by your function are bundled together in one or more files. We recommend using this method to vend dependencies whenever possible. Compared to adding dependencies to your deployment package, bundling your code results in improved performance due to the reduction in I/O operations.

**Creating and updating Node.js Lambda functions using .zip files**

Once you have created your .zip deployment package, you can use it to create a new Lambda function or update an existing one. You can deploy your .zip package using the Lambda console, the AWS Command Line Interface, and the Lambda API. You can also create and update Lambda functions using AWS Serverless Application Model (AWS SAM) and AWS CloudFormation.

The maximum size for a .zip deployment package for Lambda is 250 MB (unzipped). Note that this limit applies to the combined size of all the files you upload, including any Lambda layers.
Creating and updating functions with .zip files using the console

To create a new function, you must first create the function in the console, then upload your .zip archive. To update an existing function, open the page for your function, then follow the same procedure to add your updated .zip file.

If your .zip file is less than 50MB, you can create or update a function by uploading the file directly from your local machine. For .zip files greater than 50MB, you must upload your package to an Amazon S3 bucket first. For instructions on how to upload a file to an Amazon S3 bucket using the AWS Management Console, see Getting started with Amazon S3. To upload files using the AWS CLI, see Move objects in the AWS CLI User Guide.

Note

You cannot convert an existing container image function to use a .zip archive. You must create a new function.

To create a new function (console)

1. Open the Functions page of the Lambda console and choose Create Function.
2. Choose Author from scratch.
3. Under Basic information, do the following:
   a. For Function name, enter the name for your function.
   b. For Runtime, select the runtime you want to use.
   c. (Optional) For Architecture, choose the instruction set architecture for your function. The default architecture is x86_64. Ensure that the .zip deployment package for your function is compatible with the instruction set architecture you select.
4. (Optional) Under Permissions, expand Change default execution role. You can create a new Execution role or use an existing one.
5. Choose Create function. Lambda creates a basic 'Hello world' function using your chosen runtime.
To upload a .zip archive from your local machine (console)

1. In the Functions page of the Lambda console, choose the function you want to upload the .zip file for.
2. Select the Code tab.
3. In the Code source pane, choose Upload from.
5. To upload the .zip file, do the following:
   a. Select Upload, then select your .zip file in the file chooser.
   b. Choose Open.
   c. Choose Save.

To upload a .zip archive from an Amazon S3 bucket (console)

1. In the Functions page of the Lambda console, choose the function you want to upload a new .zip file for.
2. Select the Code tab.
3. In the Code source pane, choose Upload from.
4. Choose Amazon S3 location.
5. Paste the Amazon S3 link URL of your .zip file and choose Save.

Updating .zip file functions using the console code editor

For some functions with .zip deployment packages, you can use the Lambda console’s built-in code editor to update your function code directly. To use this feature, your function must meet the following criteria:

- Your function must use one of the interpreted language runtimes (Python, Node.js, or Ruby)
- Your function’s deployment package must be smaller than 3MB.

Function code for functions with container image deployment packages cannot be edited directly in the console.
To update function code using the console code editor

1. Open the Functions page of the Lambda console and select your function.
2. Select the Code tab.
3. In the Code source pane, select your source code file and edit it in the integrated code editor.
4. When you have finished editing your code, choose Deploy to save your changes and update your function.

Creating and updating functions with .zip files using the AWS CLI

You can use the AWS CLI to create a new function or to update an existing one using a .zip file. Use the create-function and update-function-code commands to deploy your .zip package. If your .zip file is smaller than 50MB, you can upload the .zip package from a file location on your local build machine. For larger files, you must upload your .zip package from an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the AWS CLI User Guide.

Note

If you upload your .zip file from an Amazon S3 bucket using the AWS CLI, the bucket must be located in the same AWS Region as your function.

To create a new function using a .zip file with the AWS CLI, you must specify the following:

- The name of your function (--function-name)
- Your function's runtime (--runtime)
- The Amazon Resource Name (ARN) of your function's execution role (--role)
- The name of the handler method in your function code (--handler)

You must also specify the location of your .zip file. If your .zip file is located in a folder on your local build machine, use the --zip-file option to specify the file path, as shown in the following example command.

```
aws lambda create-function --function-name myFunction \
--runtime nodejs20.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role
```
--zip-file fileb://myFunction.zip

To specify the location of .zip file in an Amazon S3 bucket, use the --code option as shown in the following example command. You only need to use the S3ObjectVersion parameter for versioned objects.

```bash
aws lambda create-function --function-name myFunction \
--runtime nodejs20.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=myBucketName,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

To update an existing function using the CLI, you specify the the name of your function using the --function-name parameter. You must also specify the location of the .zip file you want to use to update your function code. If your .zip file is located in a folder on your local build machine, use the --zip-file option to specify the file path, as shown in the following example command.

```bash
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the --s3-bucket and --s3-key options as shown in the following example command. You only need to use the --s3-object-version parameter for versioned objects.

```bash
aws lambda update-function-code --function-name myFunction \
--s3-bucket myBucketName --s3-key myFileName.zip --s3-object-version myObjectVersion
```

Creating and updating functions with .zip files using the Lambda API

To create and update functions using a .zip file archive, use the following API operations:

- [CreateFunction](#)
- [UpdateFunctionCode](#)

Creating and updating functions with .zip files using AWS SAM

The AWS Serverless Application Model (AWS SAM) is a toolkit that helps streamline the process of building and running serverless applications on AWS. You define the resources for your application in a YAML or JSON template and use the AWS SAM command line interface (AWS SAM CLI) to build,
package, and deploy your applications. When you build a Lambda function from an AWS SAM template, AWS SAM automatically creates a .zip deployment package or container image with your function code and any dependencies you specify. To learn more about using AWS SAM to build and deploy Lambda functions, see Getting started with AWS SAM in the AWS Serverless Application Model Developer Guide.

You can also use AWS SAM to create a Lambda function using an existing .zip file archive. To create a Lambda function using AWS SAM, you can save your .zip file in an Amazon S3 bucket or in a local folder on your build machine. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the AWS CLI User Guide.

In your AWS SAM template, the AWS::Serverless::Function resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

- **PackageType** - set to Zip
- **CodeUri** - set to the function code's Amazon S3 URI, path to local folder, or FunctionCode object
- **Runtime** - Set to your chosen runtime

With AWS SAM, if your .zip file is larger than 50MB, you don’t need to upload it to an Amazon S3 bucket first. AWS SAM can upload .zip packages up to the maximum allowed size of 250MB (unzipped) from a location on your local build machine.

To learn more about deploying functions using .zip file in AWS SAM, see AWS::Serverless::Function in the AWS SAM Developer Guide.

**Creating and updating functions with .zip files using AWS CloudFormation**

You can use AWS CloudFormation to create a Lambda function using a .zip file archive. To create a Lambda function from a .zip file, you must first upload your file to an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the AWS CLI User Guide.

In your AWS CloudFormation template, the AWS::Lambda::Function resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

- **PackageType** - Set to Zip
- **Code** - Enter the Amazon S3 bucket name and the .zip file name in the S3Bucket and S3Key fields
• Runtime - Set to your chosen runtime

The .zip file that AWS CloudFormation generates cannot exceed 4MB. To learn more about deploying functions using .zip file in AWS CloudFormation, see [AWS::Lambda::Function](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-lambda-function.html) in the *AWS CloudFormation User Guide*. 
Deploy Node.js Lambda functions with container images

There are three ways to build a container image for a Node.js Lambda function:

- **Using an AWS base image for Node.js**

  The AWS base images are preloaded with a language runtime, a runtime interface client to manage the interaction between Lambda and your function code, and a runtime interface emulator for local testing.

- **Using an AWS OS-only base image**

  AWS OS-only base images contain an Amazon Linux distribution and the runtime interface emulator. These images are commonly used to create container images for compiled languages, such as Go and Rust, and for a language or language version that Lambda doesn't provide a base image for, such as Node.js 19. You can also use OS-only base images to implement a custom runtime. To make the image compatible with Lambda, you must include the runtime interface client for Node.js in the image.

- **Using a non-AWS base image**

  You can use an alternative base image from another container registry, such as Alpine Linux or Debian. You can also use a custom image created by your organization. To make the image compatible with Lambda, you must include the runtime interface client for Node.js in the image.

Tip

To reduce the time it takes for Lambda container functions to become active, see Use multi-stage builds in the Docker documentation. To build efficient container images, follow the Best practices for writing Dockerfiles.

This page explains how to build, test, and deploy container images for Lambda.

**Topics**

- AWS base images for Node.js
- Using an AWS base image for Node.js
- Using an alternative base image with the runtime interface client
### AWS base images for Node.js

AWS provides the following base images for Node.js:

<table>
<thead>
<tr>
<th>Tags</th>
<th>Runtime</th>
<th>Operating system</th>
<th>Dockerfile</th>
<th>Deprecation</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Node.js 20</td>
<td>Amazon Linux 2023</td>
<td><a href="#">Dockerfile for Node.js 20 on GitHub</a></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Node.js 18</td>
<td>Amazon Linux 2</td>
<td><a href="#">Dockerfile for Node.js 18 on GitHub</a></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Node.js 16</td>
<td>Amazon Linux 2</td>
<td><a href="#">Dockerfile for Node.js 16 on GitHub</a></td>
<td>Jun 12, 2024</td>
</tr>
</tbody>
</table>

Amazon ECR repository: [gallery.ecr.aws/lambda/nodejs](#)

### Using an AWS base image for Node.js

#### Prerequisites

To complete the steps in this section, you must have the following:

- AWS Command Line Interface (AWS CLI) version 2
- Docker
- Node.js

#### Creating an image from a base image

**To create a container image from an AWS base image for Node.js**

1. Create a directory for the project, and then switch to that directory.
   ```bash
   mkdir example
   cd example
   ```
2. Create a new Node.js project with npm. To accept the default options provided in the interactive experience, press Enter.
npm init

3. Create a new file called `index.js`. You can add the following sample function code to the file for testing, or use your own.

**Example CommonJS handler**

```javascript
exports.handler = async (event) => {
  const response = {
    statusCode: 200,
    body: JSON.stringify('Hello from Lambda!'),
  };
  return response;
};
```

4. If your function depends on libraries other than the AWS SDK for JavaScript, use `npm` to add them to your package.

5. Create a new Dockerfile with the following configuration:

   - Set the `FROM` property to the [URI of the base image](https://aws.amazon.com/lambda/).  
   - Use the `COPY` command to copy the function code and runtime dependencies to `{LAMBDA_TASK_ROOT}`.  
   - Set the `CMD` argument to the Lambda function handler.

**Example Dockerfile**

```bash
FROM public.ecr.aws/lambda/nodejs:20

# Copy function code
COPY index.js ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside of the Dockerfile)
CMD [ "index.handler" ]
```

6. Build the Docker image with the `docker build` command. The following example names the image `docker-image` and gives it the test `tag`.  

Using an AWS base image
docker build --platform linux/amd64 -t docker-image:test.

### Note
The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

### (Optional) Test the image locally

1. Start the Docker image with the `docker run` command. In this example, `docker-image` is the image name and `test` is the tag.

```bash
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

### Note
If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

2. From a new terminal window, post an event to the local endpoint.

**Linux/macOS**

In Linux and macOS, run the following `curl` command:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'"'
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:
### curl

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{"payload":"hello world!"}'
```

### PowerShell

In PowerShell, run the following Invoke-WebRequest command:

```bash
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{}' -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```bash
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType "application/json"
```

3. Get the container ID.

```bash
docker ps
```

4. Use the `docker kill` command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```bash
docker kill 3766c4ab331c
```

### Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the `get-login-password` command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace `111122223333` with your AWS account ID.
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com

2. Create a repository in Amazon ECR using the `create-repository` command.

aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE

⚠️ Note
The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```json
{
  "repository": {
    "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
    "registryId": "111122223333",
    "repositoryName": "hello-world",
    "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
    "createdAt": "2023-03-09T10:39:01+00:00",
    "imageTagMutability": "MUTABLE",
    "imageScanningConfiguration": {
      "scanOnPush": true
    },
    "encryptionConfiguration": {
      "encryptionType": "AES256"
    }
  }
}
```

3. Copy the `repositoryUri` from the output in the previous step.

4. Run the `docker tag` command to tag your local image into your Amazon ECR repository as the latest version. In this command:

- Replace `docker-image:tag` with the name and `tag` of your Docker image.
• Replace `<ECRrepositoryUri>` with the repositoryUri that you copied. Make sure to include :latest at the end of the URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Example:
```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the `docker push` command to deploy your local image to the Amazon ECR repository. Make sure to include :latest at the end of the repository URI.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For ImageUri, specify the repository URI from earlier. Make sure to include :latest at the end of the URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

**Note**

You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see [Amazon ECR cross-account permissions](#).

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:
9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the `update-function-code` command to deploy the image to the Lambda function.

**Using an alternative base image with the runtime interface client**

If you use an [OS-only base image](#) or an alternative base image, you must include the runtime interface client in your image. The runtime interface client extends the [Lambda runtime API](#), which manages the interaction between Lambda and your function code.

Install the [Node.js runtime interface client](#) using the npm package manager:

```bash
npm install aws-lambda-ric
```

You can also download the [Node.js runtime interface client](#) from GitHub. The runtime interface client supports the following Node.js versions:

- 14.x
- 16.x
- 18.x
- 20.x

The following example demonstrates how to build a container image for Node.js using a non-AWS base image. The example Dockerfile uses a `buster` base image. The Dockerfile includes the runtime interface client.

**Prerequisites**

To complete the steps in this section, you must have the following:
Creating an image from an alternative base image

To create a container image from a non-AWS base image

1. Create a directory for the project, and then switch to that directory.

   ```
   mkdir example
   cd example
   ```

2. Create a new Node.js project with npm. To accept the default options provided in the interactive experience, press Enter.

   ```
   npm init
   ```

3. Create a new file called `index.js`. You can add the following sample function code to the file for testing, or use your own.

   **Example CommonJS handler**

   ```javascript
   exports.handler = async (event) => {
     const response = {
       statusCode: 200,
       body: JSON.stringify('Hello from Lambda!'),
     };
     return response;
   };
   ```

4. Create a new Dockerfile. The following Dockerfile uses a `buster` base image instead of an AWS base image. The Dockerfile includes the runtime interface client, which makes the image compatible with Lambda. The Dockerfile uses a multi-stage build. The first stage creates a build image, which is a standard Node.js environment where the function's dependencies are installed. The second stage creates a slimmer image which includes the function code and its dependencies. This reduces the final image size.

   - Set the FROM property to the base image identifier.
• Use the COPY command to copy the function code and runtime dependencies.

• Set the ENTRYPOINT to the module that you want the Docker container to run when it starts. In this case, the module is the runtime interface client.

• Set the CMD argument to the Lambda function handler.

Example Dockerfile

```bash
# Define custom function directory
ARG FUNCTION_DIR="/function"

FROM node:20-buster as build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Install build dependencies
RUN apt-get update && \
    apt-get install -y \ 
    g++ \ 
    make \ 
    cmake \ 
    unzip \ 
    libcurl4-openssl-dev

# Copy function code
RUN mkdir -p ${FUNCTION_DIR} \
    COPY . ${FUNCTION_DIR}

WORKDIR ${FUNCTION_DIR}

# Install Node.js dependencies
RUN npm install

# Install the runtime interface client
RUN npm install aws-lambda-ric

# Grab a fresh slim copy of the image to reduce the final size
FROM node:20-buster-slim

# Required for Node runtimes which use npm@8.6.0+ because 
# by default npm writes logs under /home/.npm and Lambda fs is read-only
```
ENV NPM_CONFIG_CACHE=/tmp/.npm

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT ["/usr/local/bin/npx", "aws-lambda-ric"]
# Pass the name of the function handler as an argument to the runtime
CMD ["index.handler"]

5. Build the Docker image with the `docker build` command. The following example names the image `docker-image` and gives it the test tag.

```
docker build --platform linux/amd64 -t docker-image:test .
```

Note

The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

Use the `runtime interface emulator` to locally test the image. You can build the emulator into your image or install it on your local machine.

To install and run the runtime interface emulator on your local machine

1. From your project directory, run the following command to download the runtime interface emulator (x86-64 architecture) from GitHub and install it on your local machine.
Linux/macOS

```
mkdir -p ~/.aws-lambda-rie && 
chmod +x ~/.aws-lambda-rie/aws-lambda-rie
```

To install the arm64 emulator, replace the GitHub repository URL in the previous command with the following:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
```

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

To install the arm64 emulator, replace the $downloadLink with the following:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
```

2. Start the Docker image with the `docker run` command. Note the following:

- `docker-image` is the image name and `test` is the tag.
- `/usr/local/bin/npx aws-lambda-ric index.handler` is the `ENTRYPOINT` followed by the CMD from your Dockerfile.
Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p 9000:8080 \
   --entrypoint /aws-lambda/aws-lambda-rie \
   docker-image:test \ 
   /usr/local/bin/npx aws-lambda-ric index.handler
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p 9000:8080 ` 
   --entrypoint /aws-lambda/aws-lambda-rie ` 
   docker-image:test ` 
   /usr/local/bin/npx aws-lambda-ric index.handler
```

This command runs the image as a container and creates a local endpoint at localhost:9000/2015-03-31/functions/function/invocations.

Note

If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

3. Post an event to the local endpoint.

Linux/macOS

In Linux and macOS, run the following `curl` command:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:
In PowerShell, run the following **Invoke-WebRequest** command:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{}" -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType "application/json"
```

4. Get the container ID.

```bash
docker ps
```

5. Use the **docker kill** command to stop the container. In this command, replace 3766c4ab331c with the container ID from the previous step.

```bash
docker kill 3766c4ab331c
```

### Deploying the image

#### To upload the image to Amazon ECR and create the Lambda function

1. Run the **get-login-password** command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the **--region** value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace 111122223333 with your AWS account ID.
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com

2. Create a repository in Amazon ECR using the create-repository command.

aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE

Note

The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```json
{
  "repository": {
    "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
    "registryId": "111122223333",
    "repositoryName": "hello-world",
    "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
    "createdAt": "2023-03-09T10:39:01+00:00",
    "imageTagMutability": "MUTABLE",
    "imageScanningConfiguration": {
      "scanOnPush": true
    },
    "encryptionConfiguration": {
      "encryptionType": "AES256"
    }
  }
}
```

3. Copy the repositoryUri from the output in the previous step.

4. Run the docker tag command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - Replace docker-image:test with the name and tag of your Docker image.
Replace `<ECRrepositoryUri>` with the repositoryUri that you copied. Make sure to include :latest at the end of the URI.

```bash
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Example:

```bash
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the `docker push` command to deploy your local image to the Amazon ECR repository. Make sure to include :latest at the end of the repository URI.

```bash
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For ImageUri, specify the repository URI from earlier. Make sure to include :latest at the end of the URI.

```bash
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

**Note**

You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see [Amazon ECR cross-account permissions](#).

8. Invoke the function.

```bash
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:
9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the `update-function-code` command to deploy the image to the Lambda function.

```json
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```
AWS Lambda context object in Node.js

When Lambda runs your function, it passes a context object to the handler. This object provides methods and properties that provide information about the invocation, function, and execution environment.

**Context methods**

- getRemainingTimeInMillis() – Returns the number of milliseconds left before the execution times out.

**Context properties**

- functionName – The name of the Lambda function.
- functionVersion – The version of the function.
- invokedFunctionArn – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- memoryLimitInMB – The amount of memory that's allocated for the function.
- awsRequestId – The identifier of the invocation request.
- logGroupName – The log group for the function.
- logStreamName – The log stream for the function instance.
- identity – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
  - cognitoIdentityId – The authenticated Amazon Cognito identity.
  - cognitoIdentityPoolId – The Amazon Cognito identity pool that authorized the invocation.
- clientContext – (mobile apps) Client context that's provided to Lambda by the client application.
  - client.installation_id
  - client.app_title
  - client.app_version_name
  - client.app_version_code
  - client.app_package_name
  - env.platform_version
• env.platform
• env.make
• env.model
• env.locale
• Custom – Custom values that are set by the client application.

• callbackWaitsForEmptyEventLoop – Set to false to send the response right away when
the callback runs, instead of waiting for the Node.js event loop to be empty. If this is false, any
outstanding events continue to run during the next invocation.

The following example function logs context information and returns the location of the logs.

**Example index.js file**

```javascript
exports.handler = async function(event, context) {
    console.log('Remaining time: ', context.getRemainingTimeInMillis())
    console.log('Function name: ', context.functionName)
    return context.logStreamName
}
```
AWS Lambda function logging in Node.js

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see Using Amazon CloudWatch logs with AWS Lambda.

This page describes how to produce log output from your Lambda function's code, or access logs using the AWS Command Line Interface, the Lambda console, or the CloudWatch console.

Sections
• Creating a function that returns logs
• Using Lambda advanced logging controls with Node.js
• Using the Lambda console
• Using the CloudWatch console
• Using the AWS Command Line Interface (AWS CLI)
• Deleting logs

Creating a function that returns logs

To output logs from your function code, you can use methods on the console object, or any logging library that writes to stdout or stderr. The following example logs the values of environment variables and the event object.

Example index.js file – Logging

```javascript
exports.handler = async function(event, context) {
    console.log("ENVIRONMENT VARIABLES\n" + JSON.stringify(process.env, null, 2))
    console.info("EVENT\n" + JSON.stringify(event, null, 2))
    console.warn("Event not processed.")
    return context.logStreamName
}
```

Example log format

```
START RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Version: $LATEST
```

Logging
The Node.js runtime logs the START, END, and REPORT lines for each invocation. It adds a timestamp, request ID, and log level to each entry logged by the function. The report line provides the following details.

**REPORT line data fields**

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function’s handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TraceId** – For traced requests, the [AWS X-Ray trace ID](https://aws.amazon.com/xray/).
- **SegmentId** – For traced requests, the X-Ray segment ID.
• **Sampled** – For traced requests, the sampling result.

You can view logs in the Lambda console, in the CloudWatch Logs console, or from the command line.

**Using Lambda advanced logging controls with Node.js**

To give you more control over how your functions’ logs are captured, processed, and consumed, you can configure the following logging options for supported Node.js runtimes:

- **Log format** - select between plain text and structured JSON format for your function’s logs
- **Log level** - for logs in JSON format, choose the detail level of the logs Lambda sends to Amazon CloudWatch, such as ERROR, DEBUG, or INFO
- **Log group** - choose the CloudWatch log group your function sends logs to

For more information about these logging options, and instructions on how to configure your function to use them, see the section called “Configuring advanced logging controls for your Lambda function”.

To use the log format and log level options with your Node.js Lambda functions, see the guidance in the following sections.

**Using structured JSON logs with Node.js**

If you select JSON for your function's log format, Lambda will send logs output using the console methods of `console.trace`, `console.debug`, `console.log`, `console.info`, `console.error`, and `console.warn` to CloudWatch as structured JSON. Each JSON log object contains at least four key value pairs with the following keys:

- "timestamp" - the time the log message was generated
- "level" - the log level assigned to the message
- "message" - the contents of the log message
- "requestId" - the unique request ID for the function invocation

Depending on the logging method that your function uses, this JSON object may also contain additional key pairs. For example, if your function uses console methods to log error objects
using multiple arguments, the JSON object will contain extra key value pairs with the keys `errorMessage`, `errorType`, and `stackTrace`.

If your code already uses another logging library, such as Powertools for AWS Lambda, to produce JSON structured logs, you don’t need to make any changes. Lambda doesn’t double-encode any logs that are already JSON encoded, so your function’s application logs will continue to be captured as before.

For more information about using the Powertools for AWS Lambda logging package to create JSON structured logs in the Node.js runtime, see the section called “Logging”.

**Example JSON formatted log outputs**

The following examples shows how various log outputs generated using the console methods with single and multiple arguments are captured in CloudWatch Logs when you set your function’s log format to JSON.

The first example uses the `console.error` method to output a simple string.

**Example Node.js logging code**

```javascript
export const handler = async (event) => {
  console.error("This is a warning message");
  ...
}
```

**Example JSON log record**

```json
{
  "timestamp":"2023-11-01T00:21:51.358Z",
  "level":"ERROR",
  "message":"This is a warning message",
  "requestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

You can also output more complex structured log messages using either single or multiple arguments with the console methods. In the next example, you use `console.log` to output two key value pairs using a single argument. Note that the "message" field in the JSON object Lambda sends to CloudWatch Logs is not stringified.
Example Node.js logging code

```javascript
export const handler = async (event) => {
  console.log({data: 12.3, flag: false});
  ...
}
```

Example JSON log record

```json
{
  "timestamp": "2023-12-08T23:04:664Z",
  "level": "INFO",
  "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
  "message": {
    "data": 12.3,
    "flag": false
  }
}
```

In the next example, you again use the `console.log` method to create a log output. This time, the method takes two arguments, a map containing two key value pairs and an identifying string. Note that in this case, because you have supplied two arguments, Lambda stringifies the "message" field.

Example Node.js logging code

```javascript
export const handler = async (event) => {
  console.log('Some object - ', {data: 12.3, flag: false});
  ...
}
```

Example JSON log record

```json
{
  "timestamp": "2023-12-08T23:04:664Z",
  "level": "INFO",
  "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
  "message": "Some object -  { data: 12.3, flag: false }
```

Lambda assigns outputs generated using `console.log` the log level INFO.
The final example shows how error objects can be output to CloudWatch Logs using the console methods. Note that when you log error objects using multiple arguments, Lambda adds the fields `errorMessage`, `errorType`, and `stackTrace` to the log output. To learn more about function errors in Node.js, see the section called “Errors”.

**Example Node.js logging code**

```javascript
export const handler = async (event) => {
  let e1 = new ReferenceError("some reference error");
  let e2 = new SyntaxError("some syntax error");
  console.log(e1);
  console.log("errors logged - ", e1, e2);
};
```

**Example JSON log record**

```json
{
  "timestamp": "2023-12-08T23:21:04.632Z",
  "level": "INFO",
  "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
  "message": {
    "errorType": "ReferenceError",
    "errorMessage": "some reference error",
    "stackTrace": [
      "ReferenceError: some reference error",
      "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
      "    at Runtime.handleOnceNonStreaming (file:///var/runtime/index.mjs:1173:29)"
    ]
  }
}
```

```json
{
  "timestamp": "2023-12-08T23:21:04.646Z",
  "level": "INFO",
  "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
  "message": "errors logged - ReferenceError: some reference error
    at Runtime.handler (file:///var/task/index.mjs:3:12)
    at Runtime.handleOnceNonStreaming (file:///var/runtime/index.mjs:1173:29)"
};
```
When logging multiple error types, the extra fields `errorMessage`, `errorType`, and `stackTrace` are extracted from the first error type supplied to the `console` method.

### Using embedded metric format (EMF) client libraries with structured JSON logs

AWS provides open-sourced client libraries for Node.js which you can use to create [embedded metric format](https://docs.aws.amazon.com/lambda/latest/dg/embedded-metrics.html) (EMF) logs. If you have existing functions that use these libraries and you change your function's log format to JSON, CloudWatch may no longer recognize the metrics emitted by your code.

If your code currently emits EMF logs directly using `console.log` or by using Powertools for AWS Lambda (TypeScript), CloudWatch will also be unable to parse these if you change your function's log format to JSON.

⚠️ **Important**

To ensure that your functions' EMF logs continue to be properly parsed by CloudWatch, update your [EMF](https://docs.aws.amazon.com/lambda/latest/dg/embedded-metrics.html) and [Powertools for AWS Lambda](https://github.com/aws-lambda-powertools) libraries to the latest versions. If switching to the JSON log format, we also recommend that you carry out testing to ensure compatibility with your function's embedded metrics. If your code emits EMF logs directly using `console.log`, change your code to output those metrics directly to `stdout` as shown in the following code example.

#### Example code emitting embedded metrics to `stdout`

```javascript
process.stdout.write(JSON.stringify(
  {
    "_aws": {
      "Timestamp": Date.now(),
    }
  }
));
```
Using log-level filtering with Node.js

For AWS Lambda to filter your application logs according to their log level, your function must use JSON formatted logs. You can achieve this in two ways:

- Create log outputs using the standard console methods and configure your function to use JSON log formatting. AWS Lambda then filters your log outputs using the “level” key value pair in the JSON object described in the section called “Using structured JSON logs with Node.js”. To learn how to configure your function's log format, see the section called “Configuring advanced logging controls for your Lambda function”.

- Use another logging library or method to create JSON structured logs in your code that include a “level” key value pair defining the level of the log output. For example, you can use Powertools for AWS Lambda to generate JSON structured log outputs from your code. See the section called “Logging” to learn more about using Powertools with the Node.js runtime.

For Lambda to filter your function's logs, you must also include a "timestamp" key value pair in your JSON log output. The time must be specified in valid RFC 3339 timestamp format. If you don't supply a valid timestamp, Lambda will assign the log the level INFO and add a timestamp for you.

When you configure your function to use log-level filtering, you select the level of logs you want AWS Lambda to send to CloudWatch Logs from the following options:
<table>
<thead>
<tr>
<th>Log level</th>
<th>Standard usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACE (most detail)</td>
<td>The most fine-grained information used to trace the path of your code's execution</td>
</tr>
<tr>
<td>DEBUG</td>
<td>Detailed information for system debugging</td>
</tr>
<tr>
<td>INFO</td>
<td>Messages that record the normal operation of your function</td>
</tr>
<tr>
<td>WARN</td>
<td>Messages about potential errors that may lead to unexpected behavior if unaddressed</td>
</tr>
<tr>
<td>ERROR</td>
<td>Messages about problems that prevent the code from performing as expected</td>
</tr>
<tr>
<td>FATAL (least detail)</td>
<td>Messages about serious errors that cause the application to stop functioning</td>
</tr>
</tbody>
</table>

Lambda sends logs of the selected level and lower to CloudWatch. For example, if you configure a log level of WARN, Lambda will send logs corresponding to the WARN, ERROR, and FATAL levels.

**Using the Lambda console**

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded Code editor, you will find logs in the execution results. When you use the console test feature to invoke a function, you'll find Log output in the Details section.

**Using the CloudWatch console**

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

**To view logs on the CloudWatch console**

1. Open the Log groups page on the CloudWatch console.
2. Choose the log group for your function (/aws/lambda/your-function-name).
3. Choose a log stream.

Each log stream corresponds to an **instance of your function**. A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see [Error processor sample application for AWS Lambda](#).

**Using the AWS Command Line Interface (AWS CLI)**

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface (AWS CLI) version 2](#)
- [AWS CLI – Quick configuration with `aws configure`](#)

You can use the [AWS CLI](#) to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

**Example retrieve a log ID**

The following example shows how to retrieve a log ID from the `LogResult` field for a function named `my-function`.

```bash
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```json
{
   "StatusCode": 200,
   "LogResult": "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTEzTgtOGNkYS0yOTc0YzVlNGZiMjEgUmVyc2lvb...",
   "ExecutedVersion": "$LATEST"
}
```
Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \ 
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --decode
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-aws-configure-set.html) in the AWS Command Line Interface User Guide for Version 2.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Duration: 79.67 ms     Billed Duration: 80 ms Memory Size: 128 MB Max Memory Used: 73 MB
```

The `base64` utility is available on Linux, macOS, and Ubuntu on Windows. macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

```
aws lambda invoke --function-name my-function out --log-type Tail \ 
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --decode
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-aws-configure-set.html) in the AWS Command Line Interface User Guide for Version 2.
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-name stream1 --limit 5

Example macOS and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```bash
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```bash
./get-logs.sh
```

You should see the following output:

```json
{
  "StatusCode": 200,
  "ExecutedVersion": "$LATEST"
}
{
  "events": [
    {
      "timestamp": 1559763003171,
      "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST\n",
      "ingestionTime": 1559763003309
    },
    {
      "timestamp": 1559763003173,
      "message": "2019-06-05T19:30:03.173Z	4ce9340a-b765-490f-ad8a-02ab3415e2bf INFO\tENVIRONMENT VARIABLES\r\nAWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
\r ...
",
      "ingestionTime": 1559763018353
    }
  ]
}
```
Deleting logs

Log groups aren’t deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which logs are deleted automatically.
AWS Lambda function errors in Node.js

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

This page describes how to view Lambda function invocation errors for the Node.js runtime using the Lambda console and the AWS CLI.

Sections

• Syntax
• How it works
• Using the Lambda console
• Using the AWS Command Line Interface (AWS CLI)
• Error handling in other AWS services
• What’s next?

Syntax

Example index.js file – Reference error

```
exports.handler = async function() {
    return x + 10
}
```

This code results in a reference error. Lambda catches the error and generates a JSON document with fields for the error message, the type, and the stack trace.

```
{
    "errorType": "ReferenceError",
    "errorMessage": "x is not defined",
    "trace": [
        "ReferenceError: x is not defined",
        " at Runtime.exports.handler (/var/task/index.js:2:3)",
        " at Runtime.handleOnce (/var/runtime/Runtime.js:63:25)",
        " at process._tickCallback (internal/process/next_tick.js:68:7)"
    ]
}
```
How it works

When you invoke a Lambda function, Lambda receives the invocation request and validates the permissions in your execution role, verifies that the event document is a valid JSON document, and checks parameter values.

If the request passes validation, Lambda sends the request to a function instance. The Lambda runtime environment converts the event document into an object, and passes it to your function handler.

If Lambda encounters an error, it returns an exception type, message, and HTTP status code that indicates the cause of the error. The client or service that invoked the Lambda function can handle the error programmatically, or pass it along to an end user. The correct error handling behavior depends on the type of application, the audience, and the source of the error.

The following list describes the range of status codes you can receive from Lambda.

2xx

A 2xx series error with a X-Amz-Function-Error header in the response indicates a Lambda runtime or function error. A 2xx series status code indicates that Lambda accepted the request, but instead of an error code, Lambda indicates the error by including the X-Amz-Function-Error header in the response.

4xx

A 4xx series error indicates an error that the invoking client or service can fix by modifying the request, requesting permission, or by retrying the request. 4xx series errors other than 429 generally indicate an error with the request.

5xx

A 5xx series error indicates an issue with Lambda, or an issue with the function's configuration or resources. 5xx series errors can indicate a temporary condition that can be resolved without any action by the user. These issues can't be addressed by the invoking client or service, but a Lambda function's owner may be able to fix the issue.

For a complete list of invocation errors, see InvokeFunction errors.
Using the Lambda console

You can invoke your function on the Lambda console by configuring a test event and viewing the output. The output is captured in the function's execution logs and, when active tracing is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the Functions page of the Lambda console.
2. Choose the function to test, and choose Test.
3. Under Test event, select New event.
4. Select a Template.
5. For Name, enter a name for the test. In the text entry box, enter the JSON test event.
6. Choose Save changes.
7. Choose Test.

The Lambda console invokes your function synchronously and displays the result. To see the response, logs, and other information, expand the Details section.

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- AWS Command Line Interface (AWS CLI) version 2
- AWS CLI – Quick configuration with aws configure

When you invoke a Lambda function in the AWS CLI, the AWS CLI splits the response into two documents. The AWS CLI response is displayed in your command prompt. If an error has occurred, the response contains a FunctionError field. The invocation response or error returned by the function is written to an output file. For example, output.json or output.txt.

The following invoke command example demonstrates how to invoke a function and write the invocation response to an output.txt file.

```bash
aws lambda invoke
```
The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-aws-options.html) in the *AWS Command Line Interface User Guide for Version 2*.

You should see the AWS CLI response in your command prompt:

```
{
  "StatusCode": 200,
  "FunctionError": "Unhandled",
  "ExecutedVersion": "$LATEST"
}
```

You should see the function invocation response in the `output.txt` file. In the same command prompt, you can also view the output in your command prompt using:

```
cat output.txt
```

You should see the invocation response in your command prompt.

```
{"errorType":"ReferenceError","errorMessage":"x is not defined","trace":
 ["ReferenceError: x is not defined"," at Runtime.exports.handler (/var/task/index.js:2:3)"," at Runtime.handleOnce (/var/runtime/Runtime.js:63:25)"," at process._tickCallback (internal/process/next_tick.js:68:7)"]
```

Lambda also records up to 256 KB of the error object in the function's logs. For more information, see [AWS Lambda function logging in Node.js](https://docs.aws.amazon.com/lambda/latest/dg/nodejs-logging.html).

### Error handling in other AWS services

When another AWS service invokes your function, the service chooses the invocation type and retry behavior. AWS services can invoke your function on a schedule, in response to a lifecycle event on a resource, or to serve a request from a user. Some services invoke functions asynchronously and let Lambda handle errors, while others retry or pass errors back to the user.
For example, API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502 error code. To customize the error response, you must catch errors in your code and format a response in the required format.

We recommend using AWS X-Ray to determine the source of an error and its cause. X-Ray allows you to find out which component encountered an error, and see details about the errors. The following example shows a function error that resulted in a 502 response from API Gateway.

For more information, see Instrumenting Node.js code in AWS Lambda.

**What's next?**

- Learn how to show logging events for your Lambda function on the the section called “Logging” page.
Instrumenting Node.js code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of two SDK libraries:

- AWS Distro for OpenTelemetry (ADOT) – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- AWS X-Ray SDK for Node.js – An SDK for generating and sending trace data to X-Ray.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

⚠️ Important

The X-Ray and Powertools for AWS Lambda SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs.

Sections

- Using ADOT to instrument your Node.js functions
- Using the X-Ray SDK to instrument your Node.js functions
- Activating tracing with the Lambda console
- Activating tracing with the Lambda API
- Activating tracing with AWS CloudFormation
- Interpreting an X-Ray trace
- Storing runtime dependencies in a layer (X-Ray SDK)
Using ADOT to instrument your Node.js functions

ADOT provides fully managed Lambda layers that package everything you need to collect telemetry data using the OTel SDK. By consuming this layer, you can instrument your Lambda functions without having to modify any function code. You can also configure your layer to do custom initialization of OTel. For more information, see Custom configuration for the ADOT Collector on Lambda in the ADOT documentation.

For Node.js runtimes, you can add the AWS managed Lambda layer for ADOT Javascript to automatically instrument your functions. For detailed instructions on how to add this layer, see AWS Distro for OpenTelemetry Lambda Support for JavaScript in the ADOT documentation.

Using the X-Ray SDK to instrument your Node.js functions

To record details about calls that your Lambda function makes to other resources in your application, you can also use the AWS X-Ray SDK for Node.js. To get the SDK, add the aws-xray-sdk-core package to your application's dependencies.

Example blank-nodejs/package.json

```json
{
   "name": "blank-nodejs",
   "version": "1.0.0",
   "private": true,
   "devDependencies": {
      "aws-sdk": "2.631.0",
      "jest": "25.4.0"
   },
   "dependencies": {
      "aws-xray-sdk-core": "1.1.2"
   },
   "scripts": {
      "test": "jest"
   }
}
```

To instrument AWS SDK clients, wrap the aws-sdk library with the captureAWS method.

Example blank-nodejs/function/index.js – Tracing an AWS SDK client

```javascript
const AWSXRay = require('aws-xray-sdk-core')
```
The Lambda runtime sets some environment variables to configure the X-Ray SDK. For example, Lambda sets `AWS_XRAY_CONTEXT_MISSING` to `LOG_ERROR` to avoid throwing runtime errors from the X-Ray SDK. To set a custom context missing strategy, override the environment variable in your function configuration to have no value, and then you can set the context missing strategy programmatically.

**Example Example initialization code**

```javascript
const AWSXRay = require('aws-xray-sdk-core');

// Configure the context missing strategy to do nothing
AWSXRay.setContextMissingStrategy(() => {});
```

For more information, see the section called “Environment variables”.

After you add the correct dependencies and make the necessary code changes, activate tracing in your function's configuration via the Lambda console or the API.

**Activating tracing with the Lambda console**

To toggle active tracing on your Lambda function with the console, follow these steps:

**To turn on active tracing**

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring and operations tools**.
4. Choose **Edit**.
5. Under X-Ray, toggle on **Active tracing**.
6. Choose **Save**.
Activating tracing with the Lambda API

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

The following example AWS CLI command enables active tracing on a function named `my-function`.

```bash
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Activating tracing with AWS CloudFormation

To activate tracing on an `AWS::Lambda::Function` resource in an AWS CloudFormation template, use the TracingConfig property.

Example `function-inline.yml` – Tracing configuration

```yaml
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

For an AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` resource, use the Tracing property.

Example `template.yml` – Tracing configuration

```yaml
Resources:
```

Activating tracing with the Lambda API

476
Interpreting an X-Ray trace

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's execution role. Otherwise, add the AWSXRayDaemonWriteAccess policy to the execution role.

After you've configured active tracing, you can observe specific requests through your application. The X-Ray service graph shows information about your application and all its components. The following example from the error processor sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

Note

You cannot configure the X-Ray sampling rate for your functions.
When using active tracing, Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes for the primary function from the error processor sample application.

The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named `my-function`, but one has an origin of `AWS::Lambda` and the other has origin `AWS::Lambda::Function`.

This example expands the function segment to show its three subsegments:

- **Initialization** – Represents time spent loading your function and running initialization code. This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.
You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see the AWS X-Ray SDK for Node.js in the AWS X-Ray Developer Guide.

### Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see AWS X-Ray pricing.

### Storing runtime dependencies in a layer (X-Ray SDK)

If you use the X-Ray SDK to instrument AWS SDK clients your function code, your deployment package can become quite large. To avoid uploading runtime dependencies every time you update your function code, package the X-Ray SDK in a Lambda layer.

The following example shows an AWS::Serverless::LayerVersion resource that stores the AWS X-Ray SDK for Node.js.

**Example template.yml – Dependencies layer**

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
        ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-nodejs-lib
      Description: Dependencies for the blank sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - nodejs16.x
```
With this configuration, you update the library layer only if you change your runtime dependencies. Since the function deployment package contains only your code, this can help reduce upload times.

Creating a layer for dependencies requires build changes to generate the layer archive prior to deployment. For a working example, see the blank-nodejs sample application.
Building Lambda functions with TypeScript

You can use the Node.js runtime to run TypeScript code in AWS Lambda. Because Node.js doesn't run TypeScript code natively, you must first transpile your TypeScript code into JavaScript. Then, use the JavaScript files to deploy your function code to Lambda. Your code runs in an environment that includes the AWS SDK for JavaScript, with credentials from an AWS Identity and Access Management (IAM) role that you manage.

Lambda supports the following Node.js runtimes.

**Node.js**

<table>
<thead>
<tr>
<th>Name</th>
<th>Identifier</th>
<th>SDK</th>
<th>Operating system</th>
<th>Deprecation date</th>
<th>Block function create</th>
<th>Block function update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node.js 20</td>
<td>nodejs20x</td>
<td>3.362.0</td>
<td>Amazon Linux 2023</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Node.js 18</td>
<td>nodejs18x</td>
<td>3.362.0</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Node.js 16</td>
<td>nodejs16x</td>
<td>2.1374.0</td>
<td>Amazon Linux 2</td>
<td>Jun 12, 2024</td>
<td>Jul 15, 2024</td>
<td>Aug 15, 2024</td>
</tr>
</tbody>
</table>

Setting up a TypeScript development environment

Use a local integrated development environment (IDE), text editor, or [AWS Cloud9](https://aws.amazon.com/cloud9/) to write your TypeScript function code. You can’t create TypeScript code on the Lambda console.

To transpile your TypeScript code, set up a compiler such as [esbuild](https://esbuild.github.io/) or Microsoft's TypeScript compiler (tsc), which is bundled with the TypeScript distribution. You can use the [AWS Serverless Application Model (AWS SAM)](https://aws.amazon.com/serverless) or the [AWS Cloud Development Kit (AWS CDK)](https://aws.amazon.com/cdk/) to simplify building and deploying TypeScript code. Both tools use esbuild to transpile TypeScript code into JavaScript.

When using esbuild, consider the following:
There are several TypeScript caveats.

You must configure your TypeScript transpilation settings to match the Node.js runtime that you plan to use. For more information, see Target in the esbuild documentation. For an example of a tsconfig.json file that demonstrates how to target a specific Node.js version supported by Lambda, refer to the TypeScript GitHub repository.

esbuild doesn't perform type checks. To check types, use the tsc compiler. Run tsc -noEmit or add a "noEmit" parameter to your tsconfig.json file, as shown in the following example. This configures tsc to not emit JavaScript files. After checking types, use esbuild to convert the TypeScript files into JavaScript.

Example tsconfig.json

```json
{
  "compilerOptions": {
    "target": "es2020",
    "strict": true,
    "preserveConstEnums": true,
    "noEmit": true,
    "sourceMap": false,
    "module": "commonjs",
    "moduleResolution": "node",
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "isolatedModules": true,
  },
  "exclude": ["node_modules", "**/*.test.ts"]
}
```

Topics

- AWS Lambda function handler in TypeScript
- Deploy transpiled TypeScript code in Lambda with .zip file archives
- Deploy transpiled TypeScript code in Lambda with container images
- AWS Lambda context object in TypeScript
- AWS Lambda function logging in TypeScript
- AWS Lambda function testing in TypeScript
- AWS Lambda function errors in TypeScript
• Tracing TypeScript code in AWS Lambda
AWS Lambda function handler in TypeScript

The Lambda function handler is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

Example TypeScript handler

This example function logs the contents of the event object and returns the location of the logs. Note the following:

- Before using this code in a Lambda function, you must add the @types/aws-lambda package as a development dependency. This package contains the type definitions for Lambda. When @types/aws-lambda is installed, the import statement (import ... from 'aws-lambda') imports the type definitions. It does not import the aws-lambda NPM package, which is an unrelated third-party tool. For more information, see aws-lambda in the DefinitelyTyped GitHub repository.

- The handler in this example is an ES module and must be designated as such in your package.json file or by using the .mjs file extension. For more information, see Designating a function handler as an ES module.

```typescript
import { Handler } from 'aws-lambda';

export const handler: Handler = async (event, context) => {
  console.log('EVENT: 
' + JSON.stringify(event, null, 2));
  return context.logStreamName;
};
```

The runtime passes arguments to the handler method. The first argument is the event object, which contains information from the invoker. The invoker passes this information as a JSON-formatted string when it calls Invoke, and the runtime converts it to an object. When an AWS service invokes your function, the event structure varies by service. With TypeScript, we recommend using type annotations for the event object. For more information, see Using types for the event object.

The second argument is the context object, which contains information about the invocation, function, and execution environment. In the preceding example, the function gets the name of the log stream from the context object and returns it to the invoker.
You can also use a callback argument, which is a function that you can call in non-async handlers to send a response. We recommend that you use async/await instead of callbacks. Async/await provides improved readability, error handling, and efficiency. For more information about the differences between async/await and callbacks, see Using callbacks.

Using async/await

If your code performs an asynchronous task, use the async/await pattern to make sure that the handler finishes running. Async/await is a concise and readable way to write asynchronous code in Node.js, without the need for nested callbacks or chaining promises. With async/await, you can write code that reads like synchronous code, while still being asynchronous and non-blocking.

The async keyword marks a function as asynchronous, and the await keyword pauses the execution of the function until a Promise is resolved.

Example TypeScript function – asynchronous

This example uses fetch, which is available in the nodejs18.x runtime. Note the following:

- Before using this code in a Lambda function, you must add the @types/aws-lambda package as a development dependency. This package contains the type definitions for Lambda. When @types/aws-lambda is installed, the import statement (import ... from 'aws-lambda') imports the type definitions. It does not import the aws-lambda NPM package, which is an unrelated third-party tool. For more information, see aws-lambda in the DefinitelyTyped GitHub repository.

- The handler in this example is an ES module and must be designated as such in your package.json file or by using the .mjs file extension. For more information, see see Designating a function handler as an ES module.

```typescript
import { APIGatewayProxyEvent, APIGatewayProxyResult } from 'aws-lambda';
const url = 'https://aws.amazon.com/';
export const lambdaHandler = async (event: APIGatewayProxyEvent):
  Promise<APIGatewayProxyResult> => {
    try {
        // fetch is available with Node.js 18
        const res = await fetch(url);
        return {
            statusCode: res.status,
            body: JSON.stringify({
```
message: await res.text(),
  },
  ];
} catch (err) {
  console.log(err);
  return {
    statusCode: 500,
    body: JSON.stringify({
      message: 'some error happened',
    }),
  };
}


Using callbacks

We recommend that you use async/await to declare the function handler instead of using callbacks. Async/await is a better choice for several reasons:

- **Readability**: Async/await code is easier to read and understand than callback code, which can quickly become difficult to follow and result in callback hell.

- **Debugging and error handling**: Debugging callback-based code can be difficult. The call stack can become hard to follow and errors can easily be swallowed. With async/await, you can use try/catch blocks to handle errors.

- **Efficiency**: Callbacks often require switching between different parts of the code. Async/await can reduce the number of context switches, resulting in more efficient code.

When you use callbacks in your handler, the function continues to execute until the event loop is empty or the function times out. The response isn't sent to the invoker until all event loop tasks are finished. If the function times out, an error is returned instead. You can configure the runtime to send the response immediately by setting context.callbackWaitsForEmptyEventLoop to false.

The callback function takes two arguments: an Error and a response. The response object must be compatible with JSON.stringify.

**Example TypeScript function with callback**

This sample function receives an event from Amazon API Gateway, logs the event and context objects, and then returns a response to API Gateway. Note the following:
Before using this code in a Lambda function, you must add the `@types/aws-lambda` package as a development dependency. This package contains the type definitions for Lambda. When `@types/aws-lambda` is installed, the `import` statement (`import ... from 'aws-lambda'`) imports the type definitions. It does not import the `aws-lambda` NPM package, which is an unrelated third-party tool. For more information, see `aws-lambda` in the DefinitelyTyped GitHub repository.

The handler in this example is an ES module and must be designated as such in your `package.json` file or by using the `.mjs` file extension. For more information, see see Designating a function handler as an ES module.

```javascript
import { Context, APIGatewayProxyCallback, APIGatewayEvent } from 'aws-lambda';

export const lambdaHandler = (event: APIGatewayEvent, context: Context, callback: APIGatewayProxyCallback): void => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    callback(null, {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    });
};
```

**Using types for the event object**

We recommend that you don’t use the `any` type for the handler arguments and return type because you lose the ability to check types. Instead, generate an event using the `sam local generate-event` AWS Serverless Application Model CLI command, or use an open-source definition from the `@types/aws-lambda` package.

**Generating an event using the sam local generate-event command**

1. Generate an Amazon Simple Storage Service (Amazon S3) proxy event.

   ```
   sam local generate-event s3 put >> S3PutEvent.json
   ```

2. Use the `quicktype utility` to generate type definitions from the `S3PutEvent.json` file.
3. Use the generated types in your code.

```javascript
import { S3PutEvent } from './S3PutEvent';

export const lambdaHandler = async (event: S3PutEvent): Promise<void> => {
  event.Records.map((record) => console.log(record.s3.object.key));
};
```

**Generating an event using an open-source definition from the @types/aws-lambda package**

1. Add the [@types/aws-lambda](https://www.npmjs.com/package/@types/aws-lambda) package as a development dependency.

```bash
npm install -D @types/aws-lambda
```

2. Use the types in your code.

```javascript
import { S3Event } from "aws-lambda";

export const lambdaHandler = async (event: S3Event): Promise<void> => {
  event.Records.map((record) => console.log(record.s3.object.key));
};
```
Deploy transpiled TypeScript code in Lambda with .zip file archives

Before you can deploy TypeScript code to AWS Lambda, you need to transpile it into JavaScript. This page explains three ways to build and deploy TypeScript code to Lambda with .zip file archives:

- Using AWS Serverless Application Model (AWS SAM)
- Using the AWS Cloud Development Kit (AWS CDK)
- Using the AWS Command Line Interface (AWS CLI) and esbuild

AWS SAM and AWS CDK simplify building and deploying TypeScript functions. The AWS SAM template specification provides a simple and clean syntax to describe the Lambda functions, APIs, permissions, configurations, and events that make up your serverless application. The AWS CDK lets you build reliable, scalable, cost-effective applications in the cloud with the considerable expressive power of a programming language. The AWS CDK is intended for moderately to highly experienced AWS users. Both the AWS CDK and the AWS SAM use esbuild to transpile TypeScript code into JavaScript.

Using AWS SAM to deploy TypeScript code to Lambda

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application using the AWS SAM. This application implements a basic API backend. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function is invoked. The function returns a hello world message.

Note

AWS SAM uses esbuild to create Node.js Lambda functions from TypeScript code. esbuild support is currently in public preview. During public preview, esbuild support may be subject to backwards incompatible changes.

Prerequisites

To complete the steps in this section, you must have the following:
Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.
   
   ```
   sam init --app-template hello-world-typescript --name sam-app --package-type Zip --runtime nodejs18.x
   ```

2. (Optional) The sample application includes configurations for commonly used tools, such as ESLint for code linting and Jest for unit testing. To run lint and test commands:
   
   ```
   cd sam-app/hello-world
   npm install
   npm run lint
   npm run test
   ```

3. Build the app.
   
   ```
   cd sam-app
   sam build
   ```

4. Deploy the app.
   
   ```
   sam deploy --guided
   ```

5. Follow the on-screen prompts. To accept the default options provided in the interactive experience, respond with Enter.

6. The output shows the endpoint for the REST API. Open the endpoint in a browser to test the function. You should see this response:
   
   ```
   {"message":"hello world"}
   ```

7. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.
   
   ```
   sam delete
   ```
Using the AWS CDK to deploy TypeScript code to Lambda

Follow the steps below to build and deploy a sample TypeScript application using the AWS CDK. This application implements a basic API backend. It consists of an API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function is invoked. The function returns a `hello world` message.

Prerequisites

To complete the steps in this section, you must have the following:

- [AWS CLI version 2](https://aws.amazon.com/cli/)
- [AWS CDK version 2](https://aws.amazon.com/cdk/)
- Node.js 18.x
- Either [Docker](https://www.docker.com/) or [esbuild](https://esbuild.github.io/)

Deploy a sample AWS CDK application

1. Create a project directory for your new application.

```
mkdir hello-world
cd hello-world
```

2. Initialize the app.

```
cdk init app --language typescript
```

3. Add the [@types/aws-lambda](https://www.npmjs.com/package/@types/aws-lambda) package as a development dependency. This package contains the type definitions for Lambda.

```
npm install -D @types/aws-lambda
```


5. Open `hello-world.function.ts` and add the following code to the file. This is the code for the Lambda function.
Note

The import statement imports the type definitions from `@types/aws-lambda`. It does not import the `aws-lambda` NPM package, which is an unrelated third-party tool. For more information, see `aws-lambda` in the DefinitelyTyped GitHub repository.

```typescript
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context): Promise<APIGatewayProxyResult> => {
  console.log(`Event: ${JSON.stringify(event, null, 2)}`);
  console.log(`Context: ${JSON.stringify(context, null, 2)}`);
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
};
```

6. Open `hello-world.ts` and add the following code to the file. This contains the `NodejsFunction` construct, which creates the Lambda function, and the `LambdaRestApi` construct, which creates the REST API.

```typescript
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      handler: helloFunction,
    });
  }
}
```

The `NodejsFunction` construct assumes the following by default:
- Your function handler is called handler.
- The .ts file that contains the function code (hello-world.function.ts) is in the same directory as the .ts file that contains the construct (hello-world.ts). The construct uses the construct's ID ("hello-world") and the name of the Lambda handler file ("function") to find the function code. For example, if your function code is in a file called hello-world.my-function.ts, the hello-world.ts file must reference the function code like this:

```javascript
const helloFunction = new NodejsFunction(this, 'my-function');
```

You can change this behavior and configure other esbuild parameters. For more information, see Configuring esbuild in the AWS CDK API reference.

7. Open hello-world-stack.ts. This is the code that defines your AWS CDK stack. Replace the code with the following:

```javascript
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
    constructor(scope: Construct, id: string, props?: StackProps) {
        super(scope, id, props);
        new HelloWorld(this, 'hello-world');
    }
}
```

8. from the hello-world directory containing your cdk.json file, deploy your application.

```bash
cdk deploy
```

9. The AWS CDK builds and packages the Lambda function using esbuild, and then deploys the function to the Lambda runtime. The output shows the endpoint for the REST API. Open the endpoint in a browser to test the function. You should see this response:

```json
{"message":"hello world"}
```

This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.
Using the AWS CLI and esbuild to deploy TypeScript code to Lambda

The following example demonstrates how to transpile and deploy TypeScript code to Lambda using esbuild and the AWS CLI. esbuild produces one JavaScript file with all dependencies. This is the only file that you need to add to the .zip archive.

Prerequisites

To complete the steps in this section, you must have the following:

- **AWS CLI version 2**
- Node.js 18.x
- An [execution role](#) for the Lambda function
- For Windows users, a zip file utility such as 7zip.

Deploy a sample function

1. On your local machine, create a project directory for your new function.
2. Create a new Node.js project with npm or a package manager of your choice.
   ```bash
   npm init
   ```
3. Add the [@types/aws-lambda and esbuild](#) packages as development dependencies. The @types/aws-lambda package contains the type definitions for Lambda.
   ```bash
   npm install -D @types/aws-lambda esbuild
   ```
4. Create a new file called `index.ts`. Add the following code to the new file. This is the code for the Lambda function. The function returns a hello world message. The function doesn’t create any API Gateway resources.

   ```typescript
   import { Type } from '@types/aws-lambda';
   
   function handler(event: any, context: Type.Context): any {
       return 'Hello world!';
   }
   ```

   **Note**

   The `import` statement imports the type definitions from [@types/aws-lambda](#). It does not import the aws-lambda NPM package, which is an unrelated third-party tool. For more information, see [aws-lambda](#) in the DefinitelyTyped GitHub repository.
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context): Promise<APIGatewayProxyResult> => {
  console.log(`Event: ${JSON.stringify(event, null, 2)}`);
  console.log(`Context: ${JSON.stringify(context, null, 2)}`);
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
};

5. Add a build script to the package.json file. This configures esbuild to automatically create the .zip deployment package. For more information, see Build scripts in the esbuild documentation.

**Linux and MacOS**

```
"scripts": {
  "prebuild": "rm -rf dist",
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --target=es2020 --outfile=dist/index.js",
  "postbuild": "cd dist && zip -r index.zip index.js*"
},
```

**Windows**

In this example, the "postbuild" command uses the 7zip utility to create your .zip file. Use your own preferred Windows zip utility and modify the command as necessary.

```
"scripts": {
  "prebuild": "del /q dist",
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --target=es2020 --outfile=dist/index.js",
  "postbuild": "cd dist && 7z a -tzip index.zip index.js*"
},
```

6. Build the package.
7. Create a Lambda function using the .zip deployment package. Replace the highlighted text with the Amazon Resource Name (ARN) of your execution role.

```
aws lambda create-function --function-name hello-world --runtime "nodejs18.x" --role arn:aws:iam::123456789012:role/lambda-ex --zip-file "fileb://dist/index.zip" --handler index.handler
```

8. Run a test event to confirm that the function returns the following response. If you want to invoke this function using API Gateway, create and configure a REST API.

```
{
    "statusCode": 200,
    "body": "{\"message\":\"hello world\"}"
}
```
Deploy transpiled TypeScript code in Lambda with container images

You can deploy your TypeScript code to an AWS Lambda function as a Node.js container image. AWS provides base images for Node.js to help you build the container image. These base images are preloaded with a language runtime and other components that are required to run the image on Lambda. AWS provides a Dockerfile for each of the base images to help with building your container image.

If you use a community or private enterprise base image, you must add the Node.js runtime interface client (RIC) to the base image to make it compatible with Lambda.

Lambda provides a runtime interface emulator for local testing. The AWS base images for Node.js include the runtime interface emulator. If you use an alternative base image, such as an Alpine Linux or Debian image, you can build the emulator into your image or install it on your local machine.

Using a Node.js base image to build and package TypeScript function code

Prerequisites

To complete the steps in this section, you must have the following:

- AWS Command Line Interface (AWS CLI) version 2
- Docker
- Node.js 18.x

Creating an image from a base image

To create an image from an AWS base image for Lambda

1. On your local machine, create a project directory for your new function.
2. Create a new Node.js project with npm or a package manager of your choice.

   `npm init`
3. Add the `@types/aws-lambda` and `esbuild` packages as development dependencies. The `@types/aws-lambda` package contains the type definitions for Lambda.

```bash
npm install -D @types/aws-lambda esbuild
```

4. Add a **build script** to the `package.json` file.

```json
"scripts": {
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --target=es2020 --outfile=dist/index.js"
}
```

5. Create a new file called `index.ts`. Add the following sample code to the new file. This is the code for the Lambda function. The function returns a `hello world` message.

```typescript
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context): Promise<APIGatewayProxyResult> => {
  console.log(`Event: ${JSON.stringify(event, null, 2)}`);
  console.log(`Context: ${JSON.stringify(context, null, 2)}`);
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
};
```

6. Create a new Dockerfile with the following configuration:

- Set the `FROM` property to the URI of the base image.
- Set the `CMD` argument to specify the Lambda function handler.
Example Dockerfile

The following Dockerfile uses a multi-stage build. The first step transpiles the TypeScript code into JavaScript. The second step produces a container image that contains only JavaScript files and production dependencies.

```
FROM public.ecr.aws/lambda/nodejs:18 as builder
WORKDIR /usr/app
COPY package.json index.ts ./
RUN npm install
RUN npm run build

FROM public.ecr.aws/lambda/nodejs:18
WORKDIR ${LAMBDA_TASK_ROOT}
COPY --from=builder /usr/app/dist/* ./
CMD ["index.handler"]
```

7. Build the Docker image with the `docker build` command. The following example names the image `docker-image` and gives it the test tag.

```
docker build --platform linux/amd64 -t docker-image:test .
```

Note

The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

1. Start the Docker image with the `docker run` command. In this example, `docker-image` is the image name and `test` is the tag.

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```
This command runs the image as a container and creates a local endpoint at localhost:9000/2015-03-31/functions/function/invocations.

Note
If you built the Docker image for the ARM64 instruction set architecture, be sure to use the --platform linux/arm64 option instead of --platform linux/amd64.

2. From a new terminal window, post an event to the local endpoint.

Linux/macOS

In Linux and macOS, run the following curl command:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{"payload":"hello world!"}'
```

PowerShell

In PowerShell, run the following Invoke-WebRequest command:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{}' -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType "application/json"
```

3. Get the container ID.
4. Use the `docker kill` command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```bash
docker kill 3766c4ab331c
```

## Deploying the image

### To upload the image to Amazon ECR and create the Lambda function

1. Run the `get-login-password` command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace `111122223333` with your AWS account ID.

```bash
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a repository in Amazon ECR using the `create-repository` command.

```bash
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

**Note**

The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```json
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
```
3. Copy the `repositoryUri` from the output in the previous step.

4. Run the `docker tag` command to tag your local image into your Amazon ECR repository as the latest version. In this command:
   - Replace `docker-image:test` with the name and _tag_ of your Docker image.
   - Replace `<ECRrepositoryUri>` with the `repositoryUri` that you copied. Make sure to include :latest at the end of the URI.

   ```bash
docker tag docker-image:test <ECRrepositoryUri>:latest
   
   Example:

   ```bash
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. Run the `docker push` command to deploy your local image to the Amazon ECR repository. Make sure to include :latest at the end of the repository URI.

   ```bash
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. **Create an execution role** for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include :latest at the end of the URI.
aws lambda create-function
--function-name hello-world
--package-type Image
--code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
--role arn:aws:iam::111122223333:role/lambda-ex

Note
You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

aws lambda invoke --function-name hello-world response.json

You should see a response like this:

{
   "ExecutedVersion": "$LATEST",
   "StatusCode": 200
}

9. To see the output of the function, check the response.json file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the update-function-code command to deploy the image to the Lambda function.
AWS Lambda context object in TypeScript

When Lambda runs your function, it passes a context object to the handler. This object provides methods and properties that provide information about the invocation, function, and execution environment.

**Context methods**

- `getRemainingTimeInMillis()` – Returns the number of milliseconds left before the execution times out.

**Context properties**

- `functionName` – The name of the Lambda function.
- `functionVersion` – The version of the function.
- `invokedFunctionArn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memoryLimitInMB` – The amount of memory that's allocated for the function.
- `awsRequestId` – The identifier of the invocation request.
- `logGroupName` – The log group for the function.
- `logStreamName` – The log stream for the function instance.
- `identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
  - `cognitoIdentityId` – The authenticated Amazon Cognito identity.
  - `cognitoIdentityPoolId` – The Amazon Cognito identity pool that authorized the invocation.
- `clientContext` – (mobile apps) Client context that's provided to Lambda by the client application.
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `env.platform_version`
- `env.platform`
- `env.make`
- `env.model`
- `env.locale`
- Custom – Custom values that are set by the client application.

- `callbackWaitsForEmptyEventLoop` – Set to false to send the response right away when the `callback` runs, instead of waiting for the Node.js event loop to be empty. If this is false, any outstanding events continue to run during the next invocation.

You can use the [@types/aws-lambda](https://www.npmjs.com/package/@types/aws-lambda) npm package to work with the context object.

### Example index.ts file

The following example function logs context information and returns the location of the logs.

```typescript
import { Context } from 'aws-lambda';
export const lambdaHandler = async (event: string, context: Context): Promise<string> => {
  console.log('Remaining time: ', context.getRemainingTimeInMillis());
  console.log('Function name: ', context.functionName);
  return context.logStreamName;
};
```

**Note**

Before using this code in a Lambda function, you must add the [@types/aws-lambda](https://www.npmjs.com/package/@types/aws-lambda) package as a development dependency. This package contains the type definitions for Lambda. When `@types/aws-lambda` is installed, the `import` statement (import ... from 'aws-lambda') imports the type definitions. It does not import the aws-lambda NPM package, which is an unrelated third-party tool. For more information, see [aws-lambda](https://www.npmjs.com/package/aws-lambda) in the DefinitelyTyped GitHub repository.
AWS Lambda function logging in TypeScript

AWS Lambda automatically monitors Lambda functions and sends log entries to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation and other output from your function’s code to the log stream. For more information about CloudWatch Logs, see Using Amazon CloudWatch logs with AWS Lambda.

To output logs from your function code, you can use methods on the console object. For more detailed logging, you can use any logging library that writes to stdout or stderr.

Sections

- Tools and libraries
- Using Powertools for AWS Lambda (TypeScript) and AWS SAM for structured logging
- Using Powertools for AWS Lambda (TypeScript) and the AWS CDK for structured logging
- Using the Lambda console
- Using the CloudWatch console

Tools and libraries

Powertools for AWS Lambda (TypeScript) is a developer toolkit to implement Serverless best practices and increase developer velocity. The Logger utility provides a Lambda optimized logger which includes additional information about function context across all your functions with output structured as JSON. Use this utility to do the following:

- Capture key fields from the Lambda context, cold start and structures logging output as JSON
- Log Lambda invocation events when instructed (disabled by default)
- Print all the logs only for a percentage of invocations via log sampling (disabled by default)
- Append additional keys to structured log at any point in time
- Use a custom log formatter (Bring Your Own Formatter) to output logs in a structure compatible with your organization’s Logging RFC
Using Powertools for AWS Lambda (TypeScript) and AWS SAM for structured logging

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application with integrated Powertools for AWS Lambda (TypeScript) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Node.js 18.x or later
- AWS CLI version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.

   ```bash
   sam init --app-template hello-world-powertoolsTYPESCRIPT --name sam-app --package-type Zip --runtime nodejs18.x
   ```

2. Build the app.

   ```bash
   cd sam-app && sam build
   ```

3. Deploy the app.

   ```bash
   sam deploy --guided
   ```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.

   ```bash
   ```
Note
For **HelloWorld**Function may not have authorization defined, Is this okay?, make sure to enter y.

5. Get the URL of the deployed application:

```bash
aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```bash
curl <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```json
{"message":"hello world"}
```

7. To get the logs for the function, run `sam logs`. For more information, see [Working with logs](#) in the AWS Serverless Application Model Developer Guide.

```bash
sam logs --stack-name sam-app
```

The log output looks like this:

```json
2023/01/31/[$LATEST]4d53e8d279824834a1cccd35511a4949c 2022-08-31T09:33:10.552000
START RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Version: $LATEST
2023/01/31/[$LATEST]4d53e8d279824834a1cccd35511a4949c 2022-08-31T09:33:10.594000
2022-08-31T09:33:10.557Z 70693159-7e94-4102-a2af-98a6343fb8fb
INFO {"_aws":{"Timestamp":1661938390556,"CloudWatchMetrics":
["Name":"ColdStart","Unit":"Count"]},"service":"helloWorld","ColdStart":1}
2023/01/31/[$LATEST]4d53e8d279824834a1cccd35511a4949c 2022-08-31T09:33:10.595000
2022-08-31T09:33:10.595Z 70693159-7e94-4102-a2af-98a6343fb8fb
INFO {"_aws":{"Timestamp":1661938390556,"CloudWatchMetrics":
["Name":"ColdStart","Unit":"Count"]},"service":"helloWorld","ColdStart":1}
2023/01/31/[$LATEST]4d53e8d279824834a1cccd35511a4949c 2022-08-31T09:33:10.595000
2022-08-31T09:33:10.595Z 70693159-7e94-4102-a2af-98a6343fb8fb
INFO {"_aws":{"Timestamp":1661938390556,"CloudWatchMetrics":
["Name":"ColdStart","Unit":"Count"]},"service":"helloWorld","ColdStart":1}
2023/01/31/[$LATEST]4d53e8d279824834a1cccd35511a4949c 2022-08-31T09:33:10.595000
2022-08-31T09:33:10.595Z 70693159-7e94-4102-a2af-98a6343fb8fb
INFO {"_aws":{"Timestamp":1661938390556,"CloudWatchMetrics":
["Name":"ColdStart","Unit":"Count"]},"service":"helloWorld","ColdStart":1}
2023/01/31/[$LATEST]4d53e8d279824834a1cccd35511a4949c 2022-08-31T09:33:10.595000
2022-08-31T09:33:10.595Z 70693159-7e94-4102-a2af-98a6343fb8fb
INFO {"_aws":{"Timestamp":1661938390556,"CloudWatchMetrics":
["Name":"ColdStart","Unit":"Count"]},"service":"helloWorld","ColdStart":1}
```

Using Powertools for AWS Lambda (TypeScript) and AWS SAM for structured logging

508
8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

**Managing log retention**

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which CloudWatch automatically deletes the logs. To set up log retention, add the following to your AWS SAM template:

```
Resources:
    HelloWorldFunction:
        Type: AWS::Serverless::Function
        Properties:
            # Omitting other properties

    LogGroup:
        Type: AWS::Logs::LogGroup
        Properties:
            LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
            RetentionInDays: 7
```

**Using Powertools for AWS Lambda (TypeScript) and the AWS CDK for structured logging**

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application with integrated Powertools for AWS Lambda (TypeScript) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send
a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a `hello world` message.

**Prerequisites**

To complete the steps in this section, you must have the following:

- Node.js 18.x or later
- AWS CLI version 2
- AWS CDK version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

**Deploy a sample AWS CDK application**

1. Create a project directory for your new application.
   ```bash
   mkdir hello-world
cd hello-world
   ```

2. Initialize the app.
   ```bash
cdk init app --language typescript
   ```

3. Add the [@types/aws-lambda](#) package as a development dependency.
   ```bash
   npm install -D @types/aws-lambda
   ```

4. Install the Powertools [Logger utility](#).
   ```bash
   npm install @aws-lambda-powertools/logger
   ```


6. Open `hello-world.function.ts` and add the following code to the file. This is the code for the Lambda function.
   ```typescript
   import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
   ```
import { Logger } from '@aws-lambda-powertools/logger';
const logger = new Logger();

export const handler = async (event: APIGatewayEvent, context: Context):
Promise<APIGatewayProxyResult> => {
  logger.info('This is an INFO log - sending HTTP 200 - hello world response');
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
};

7. Open hello-world.ts and add the following code to the file. This contains the NodejsFunction construct, which creates the Lambda function, configures environment variables for Powertools, and sets log retention to one week. It also includes the LambdaRestApi construct, which creates the REST API.

```typescript
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { RetentionDays } from 'aws-cdk-lib/aws-logs';
import { CfnOutput } from 'aws-cdk-lib';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        Powertools_SERVICE_NAME: 'helloWorld',
        LOG_LEVEL: 'INFO',
      },
      logRetention: RetentionDays.ONE_WEEK,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
  }
```
8. Open **hello-world-stack.ts**. This is the code that defines your AWS CDK stack. Replace the code with the following:

```typescript
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
    constructor(scope: Construct, id: string, props?: StackProps) {
        super(scope, id, props);
        new HelloWorld(this, 'hello-world');
    }
}
```

9. Go back to the project directory.

   `cd hello-world`

10. Deploy your application.

    `cdk deploy`

11. Get the URL of the deployed application:

    `aws cloudformation describe-stacks --stack-name HelloWorldStack --query 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text`

12. Invoke the API endpoint:

    `curl <URL_FROM_PREVIOUS_STEP>`

If successful, you'll see this response:

```json
{"message":"hello world"}
```

13. To get the logs for the function, run **sam logs**. For more information, see Working with logs in the AWS Serverless Application Model Developer Guide.

    `sam logs --stack-name HelloWorldStack`
The log output looks like this:

```
2023/01/31/$LATEST$ 2022-08-31T14:48:37.047000
START RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Version: $LATEST
2023/01/31/$LATEST$ 2022-08-31T14:48:37.050000 {
    "level": "INFO",
    "message": "This is an INFO log - sending HTTP 200 - hello world response",
    "service": "helloWorld",
    "timestamp": "2022-08-31T14:48:37.048Z",
    "xray_trace_id": "1-630f74c4-2b080cf77680a04f2362bcf2"
}
2023/01/31/$LATEST$ 2022-08-31T14:48:37.082000 END
Requestid: 19ad1007-ff67-40ce-9afe-0af0a9eb512c
2023/01/31/$LATEST$ 2022-08-31T14:48:37.082000
REPORT RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Duration: 34.60 ms Billed Duration: 35 ms Memory Size: 128 MB Max Memory Used: 57 MB Init Duration: 173.48 ms
```

14. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

cdk destroy

**Using the Lambda console**

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded Code editor, you will find logs in the execution results. When you use the console test feature to invoke a function, you'll find Log output in the Details section.

**Using the CloudWatch console**

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

**To view logs on the CloudWatch console**

1. Open the Log groups page on the CloudWatch console.
2. Choose the log group for your function (/aws/lambda/your-function-name).
3. Choose a log stream.

Each log stream corresponds to an instance of your function. A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see Error processor sample application for AWS Lambda.
AWS Lambda function testing in TypeScript

Note
See the Testing functions chapter for a complete introduction to techniques and best practices for testing serverless solutions.

Testing serverless functions uses traditional test types and techniques, but you must also consider testing serverless applications as a whole. Cloud-based tests will provide the most accurate measure of quality of both your functions and serverless applications.

A serverless application architecture includes managed services that provide critical application functionality through API calls. For this reason, your development cycle should include automated tests that verify functionality when your function and services interact.

If you do not create cloud-based tests, you could encounter issues due to differences between your local environment and the deployed environment. Your continuous integration process should run tests against a suite of resources provisioned in the cloud before promoting your code to the next deployment environment, such as QA, Staging, or Production.

Continue reading this short guide to learn about testing strategies for serverless applications, or visit the Serverless Test Samples repository to dive in with practical examples, specific to your chosen language and runtime.
For serverless testing, you will still write *unit, integration and end-to-end* tests.

- **Unit tests** - Tests that run against an isolated block of code. For example, verifying the business logic to calculate the delivery charge given a particular item and destination.
- **Integration tests** - Tests involving two or more components or services that interact, typically in a cloud environment. For example, verifying a function processes events from a queue.
- **End-to-end tests** - Tests that verify behavior across an entire application. For example, ensuring infrastructure is set up correctly and that events flow between services as expected to record a customer's order.

**Testing your serverless applications**

You will generally use a mix of approaches to test your serverless application code, including testing in the cloud, testing with mocks, and occasionally testing with emulators.

**Testing in the cloud**

Testing in the cloud is valuable for all phases of testing, including unit tests, integration tests, and end-to-end tests. You run tests against code deployed in the cloud and interacting with cloud-based services. This approach provides the **most accurate** measure of quality of your code.

A convenient way to debug your Lambda function in the cloud is through the console with a test event. A *test event* is a JSON input to your function. If your function does not require input, the event can be an empty JSON document `{}`. The console provides sample events for a variety of service integrations. After creating an event in the console, you can share it with your team to make testing easier and consistent.

⚠️ **Note**

Testing a function in the console is a quick way to get started, but automating your test cycles ensures application quality and development speed.

**Testing tools**

Tools and techniques exist to accelerate development feedback loops. For example, AWS SAM Accelerate and AWS CDK watch mode both decrease the time required to update cloud environments.
TypeScript code uses `import` or `require` statements to import dependencies. You can use the module loading behavior of the Node.js runtime to replace or wrap dependencies before they are imported by the System Under Test (SUT).

The [AWS SDK V3 Client Mock](#) is a TypeScript library for mocking AWS services and resources. This library provides a way to mock the sending of Commands and a way to define the returned results depending on the Command type and payload.

Another commonly used framework for testing, [Jest](#), provides a custom resolver for imports so you can create mock objects that are outside of your test's scope.

For more information on mocking, read the blog post: [Unit Testing Lambda with TypeScript and Mock AWS Services](#).

To reduce the latency involved with cloud deployment iterations, see [AWS Serverless Application Model (AWS SAM) Accelerate](#), [AWS Cloud Development Kit (AWS CDK) watch mode](#). These tools monitor your infrastructure and code for changes. They react to these changes by creating and deploying incremental updates automatically into your cloud environment.

Examples that use these tools are available in the [TypeScript Test Samples](#) code repository.
AWS Lambda function errors in TypeScript

If an exception occurs in TypeScript code that's transpiled into JavaScript, use source map files to determine where the error occurred. Source map files allow debuggers to map compiled JavaScript files to the TypeScript source code.

For example, the following code results in an error:

```typescript
export const handler = async (event: unknown): Promise<unknown> => {
  throw new Error('Some exception');
};
```

AWS Lambda catches the error and generates a JSON document. However, this JSON document refers to the compiled JavaScript file (`app.js`), not the TypeScript source file.

```
{
  "errorType": "Error",
  "errorMessage": "Some exception",
  "stack": [
    "Error: Some exception",
    " at Runtime.p [as handler] (/var/task/app.js:1:491)",
    " at Runtime.handleOnce (/var/runtime/Runtime.js:66:25)"
  ]
}
```

To get an error response that maps to your TypeScript source file

1. Generate a source map file with esbuild or another TypeScript compiler. Example:
   ```
esbuild app.ts --sourcemap --outfile=output.js
   ``
2. Add the source map to your deployment.
3. Turn on source maps for the Node.js runtime by adding `--enable-source-maps` to your `NODE_OPTIONS`.

Note
The following steps aren't valid for Lambda@Edge functions because Lambda@Edge doesn't support environment variables.
Example for the AWS Serverless Application Model (AWS SAM)

Globals:
Function:
  Environment:
  Variables:
    NODE_OPTIONS: '--enable-source-maps'

Make sure that the esbuild properties in your template.yaml file include Sourcemap: true. Example:

Metadata: # Manage esbuild properties
  BuildMethod: esbuild
  BuildProperties:
    Minify: true
    Target: "es2020"
    Sourcemap: true
  EntryPoints:
    - app.ts

Example for the AWS Cloud Development Kit (AWS CDK)

To use a source map with an AWS CDK application, add the following code to the file that contains the NodejsFunction construct.

```
const helloFunction = new NodejsFunction(this, 'function',{
  bundling: {
    minify: true,
    sourceMap: true
  },
  environment:{
    NODE_OPTIONS: '--enable-source-maps',
  }
});
```

When you use a source map in your code, you get an error response similar to the following. This response shows that the error happened at line 2, column 11 in the app.ts file.

```
{
  "errorType": "Error",
  "errorMessage": "Some exception",
}
"stack": [
    "Error: Some exception",
    "    at Runtime.p (/private/var/folders/3c/0d4wz7dn2y75bw_hxdwc0h6w0000gr/T/tmpfmxb4ziy/app.ts:2:11)",
    "    at Runtime.handleOnce (/var/runtime/Runtime.js:66:25)"
]
Tracing TypeScript code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of three SDK libraries:

- **AWS Distro for OpenTelemetry (ADOT)** – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- **AWS X-Ray SDK for Node.js** – An SDK for generating and sending trace data to X-Ray.
- **Powertools for AWS Lambda (TypeScript)** – A developer toolkit to implement Serverless best practices and increase developer velocity.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

⚠️ **Important**

The X-Ray and Powertools for AWS Lambda SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see [Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs](#).

Sections

- [Using Powertools for AWS Lambda (TypeScript) and AWS SAM for tracing](#)
- [Using Powertools for AWS Lambda (TypeScript) and the AWS CDK for tracing](#)
- [Interpreting an X-Ray trace](#)
Using Powertools for AWS Lambda (TypeScript) and AWS SAM for tracing

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application with integrated Powertools for AWS Lambda (TypeScript) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a `hello world` message.

Prerequisites

To complete the steps in this section, you must have the following:

- Node.js 18.x or later
- AWS CLI version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.

   ```bash
   sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs18.x --no-tracing
   ```

2. Build the app.

   ```bash
   cd sam-app && sam build
   ```

3. Deploy the app.

   ```bash
   sam deploy --guided
   ```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.
5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message":"hello world"}
```

7. To get the traces for the function, run `sam traces`.

```
sam traces
```

The trace output looks like this:

```
XRay Event [revision 1] at (2023-01-31T11:29:40.527000) with id (1-11a2222-111a222222cb33de3b95daf9) and duration (0.483s)
- 0.425s - sam-app/Prod [HTTP: 200]
  - 0.422s - Lambda [HTTP: 200]
- 0.406s - sam-app-HelloWorldFunction-XYZv11a1bcde [HTTP: 200]
- 0.172s - sam-app-HelloWorldFunction-XYZv11a1bcde
  - 0.179s - Initialization
  - 0.112s - Invocation
    - 0.052s - ## app.lambdaHandler
      - 0.001s - ### MySubSegment
    - 0.059s - Overhead
```

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.
X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

⚠️ Note
You cannot configure the X-Ray sampling rate for your functions.

Using Powertools for AWS Lambda (TypeScript) and the AWS CDK for tracing

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application with integrated Powertools for AWS Lambda (TypeScript) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a `hello world` message.

Prerequisites

To complete the steps in this section, you must have the following:

- Node.js 18.x or later
- **AWS CLI version 2**
- **AWS CDK version 2**
- **AWS SAM CLI version 1.75 or later**. If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS Cloud Development Kit (AWS CDK) application

1. Create a project directory for your new application.
2. Initialize the app.

```bash
cdk init app --language typescript
```

3. Add the `@types/aws-lambda` package as a development dependency.

```bash
npm install -D @types/aws-lambda
```

4. Install the Powertools `Tracer utility`.

```bash
npm install @aws-lambda-powertools/tracer
```


6. Open `hello-world.function.ts` and add the following code to the file. This is the code for the Lambda function.

```typescript
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
import { Tracer } from '@aws-lambda-powertools/tracer';
const tracer = new Tracer();

export const handler = async (event: APIGatewayEvent, context: Context): Promise<APIGatewayProxyResult> => {
  // Get facade segment created by Lambda
  const segment = tracer.getSegment();

  // Create subsegment for the function and set it as active
  const handlerSegment = segment.addNewSubsegment(`## ${process.env._HANDLER}`);
  tracer.setSegment(handlerSegment);

  // Annotate subsegment with the cold start and serviceName
  tracer.annotateColdStart();
  tracer.addServiceNameAnnotation();

  // Add annotation for the awsRequestId
  tracer.putAnnotation('awsRequestId', context.awsRequestId);

  // Create another subsegment and set it as active
  const subsegment = handlerSegment.addNewSubsegment('### MySubSegment');

```
tracer.setSegment(subsegment);
let response: APIGatewayProxyResult = {
    statusCode: 200,
    body: JSON.stringify({
        message: 'hello world',
    }),
};
// Close subsegments (the Lambda one is closed automatically)
subsegment.close(); // (### MySubSegment)
handlerSegment.close(); // (## index.handler)

// Set the facade segment as active again (the one created by Lambda)
tracer.setSegment(segment);
return response;

7. Open hello-world.ts and add the following code to the file. This contains the NodejsFunction construct, which creates the Lambda function, configures environment variables for Powertools, and sets log retention to one week. It also includes the LambdaRestApi construct, which creates the REST API.

```typescript
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { CfnOutput } from 'aws-cdk-lib';
import { Tracing } from 'aws-cdk-lib/aws-lambda';

export class HelloWorld extends Construct {
    constructor(scope: Construct, id: string) {
        super(scope, id);
        const helloFunction = new NodejsFunction(this, 'function', {
            environment: {
                POWERTOOLS_SERVICE_NAME: 'helloWorld',
            },
            tracing: Tracing.ACTIVE,
        });
        const api = new LambdaRestApi(this, 'apigw', {
            handler: helloFunction,
        });
        new CfnOutput(this, 'apiUrl', {
            exportName: 'apiUrl',
            value: api.url,
        });
    }
}
```
8. Open **hello-world-stack.ts**. This is the code that defines your AWS CDK stack. Replace the code with the following:

```typescript
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
    constructor(scope: Construct, id: string, props?: StackProps) {
        super(scope, id, props);
        new HelloWorld(this, 'hello-world');
    }
}
```

9. **Deploy your application.**

   cd ..
   cdk deploy

10. **Get the URL of the deployed application:**

    ```bash
    aws cloudformation describe-stacks --stack-name HelloWorldStack --query 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
    ```

11. **Invoke the API endpoint:**

    ```bash
curl <URL_FROM_PREVIOUS_STEP>
    ```

   If successful, you'll see this response:

   ```json
   {"message":"hello world"}
   ```

12. **To get the traces for the function, run** `sam traces`

    ```bash
    sam traces
    ```

   The trace output looks like this:
13. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```bash
cdk destroy
```

## Interpreting an X-Ray trace

After you've configured active tracing, you can observe specific requests through your application. The [X-Ray trace map](#) provides information about your application and all its components. The following example shows a trace from the sample application:
You can run Python code in AWS Lambda. Lambda provides runtimes for Python that run your code to process events. Your code runs in an environment that includes the SDK for Python (Boto3), with credentials from an AWS Identity and Access Management (IAM) role that you manage.

Lambda supports the following Python runtimes.

### Python

<table>
<thead>
<tr>
<th>Name</th>
<th>Identifier</th>
<th>SDK</th>
<th>Operating system</th>
<th>Deprecation date</th>
<th>Block function create</th>
<th>Block function update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python 3.12</td>
<td>python3.12 2</td>
<td>boto3-1.2 8.83 botoocore-1.31.83</td>
<td>Amazon Linux 2023</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python 3.11</td>
<td>python3.11 1</td>
<td>boto3-1.2 8.83 botoocore-1.31.83</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python 3.10</td>
<td>python3.10 0</td>
<td>boto3-1.2 6.90 botoocore-1.29.90</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python 3.9</td>
<td>python3.9 6.90 botoocore-1.29.90</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python 3.8</td>
<td>python3.8 6.90 botoocore-1.29.90</td>
<td>Amazon Linux 2</td>
<td>Oct 14, 2024</td>
<td>Nov 13, 2024</td>
<td>Jan 7, 2025</td>
<td></td>
</tr>
</tbody>
</table>
Python 3.12 and later Python runtimes are based on the Amazon Linux 2023 (provided.al2023) runtime, which uses the AL2023 minimal container image. The Python 3.8-3.11 runtimes are based on the Amazon Linux 2 (provided.al2) runtime. AL2023 provides several advantages over Amazon Linux 2, including a smaller deployment footprint and updated versions of libraries such as glibc.

AL2023 uses microdnf (symlinked as dnf) as the package manager instead of yum, which is the default package manager in previous runtimes. microdnf is a standalone implementation of dnf. If you deploy your Lambda functions as container images, you must update your Dockerfiles to use dnf instead of yum when upgrading to the Python 3.12 base image.

For a list of packages that are included in AL2023, refer to the Minimal Container columns in Comparing packages installed on Amazon Linux 2023 Container Images. For more information about the differences between python3.12 and previous runtimes, see Python 3.12 runtime now available in AWS Lambda on the AWS Compute Blog.

To create a Python function

1. Open the Lambda console.
2. Choose Create function.
3. Configure the following settings:
   - Function name: Enter a name for the function.
   - Runtime: Choose Python 3.12.
4. Choose Create function.
5. To configure a test event, choose Test.
6. For Event name, enter test.
7. Choose Save changes.
8. To invoke the function, choose Test.
The console creates a Lambda function with a single source file named `lambda_function`. You can edit this file and add more files in the built-in code editor. To save your changes, choose Save. Then, to run your code, choose Test.

**Note**

The Lambda console uses AWS Cloud9 to provide an integrated development environment in the browser. You can also use AWS Cloud9 to develop Lambda functions in your own environment. For more information, see Working with AWS Lambda functions using the AWS Toolkit in the AWS Cloud9 user guide.

**Note**

To get started with application development in your local environment, deploy one of the sample applications available in this guide's GitHub repository.

**Sample Lambda applications in Python**

- [blank-python](#) – A Python function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.

Your Lambda function comes with a CloudWatch Logs log group. The function runtime sends details about each invocation to CloudWatch Logs. It relays any logs that your function outputs during invocation. If your function returns an error, Lambda formats the error and returns it to the invoker.

**Response format**

In Python 3.12 and later Python runtimes, functions return Unicode characters as part of their JSON response. Earlier Python runtimes return escaped sequences for Unicode characters in responses. For example, in Python 3.11, if you return a Unicode string such as "こんにちは", it escapes the Unicode characters and returns "\u3053\u3093\u306b\u3061\u306f". The Python 3.12 runtime returns the original "こんにちは".
Using Unicode responses reduces the size of Lambda responses, making it easier to fit larger responses into the 6 MB maximum payload size for synchronous functions. In the previous example, the escaped version is 32 bytes—compared to 17 bytes with the Unicode string.

When you upgrade to Python 3.12, you might need to adjust your code to account for the new response format. If the caller expects escaped Unicode, you must either add code to the returning function to escape the Unicode manually, or adjust the caller to handle the Unicode return.

**Graceful shutdown for extensions**

Python 3.12 and later Python runtimes offer improved graceful shutdown capabilities for functions with external extensions. When Lambda shuts down an execution environment, it sends a SIGTERM signal to the runtime and then a SHUTDOWN event to each registered external extension. You can catch the SIGTERM signal in your Lambda function and clean up resources such as database connections that were created by the function.

To learn more about the execution environment lifecycle, see [Lambda execution environment](#). For examples of how to use graceful shutdown with extensions, see the [AWS Samples GitHub repository](#).

**Topics**

- [Lambda function handler in Python](#)
- [Working with .zip file archives for Python Lambda functions](#)
- [Deploy Python Lambda functions with container images](#)
- [AWS Lambda context object in Python](#)
- [AWS Lambda function logging in Python](#)
- [AWS Lambda function testing in Python](#)
- [AWS Lambda function errors in Python](#)
- [Instrumenting Python code in AWS Lambda](#)
Lambda function handler in Python

The Lambda function handler is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

You can use the following general syntax when creating a function handler in Python:

```python
def handler_name(event, context):
    ...
    return some_value
```

Naming

The Lambda function handler name specified at the time that you create a Lambda function is derived from:

- The name of the file in which the Lambda handler function is located.
- The name of the Python handler function.

A function handler can be any name; however, the default name in the Lambda console is `lambda_function.lambda_handler`. This function handler name reflects the function name (`lambda_handler`) and the file where the handler code is stored (`lambda_function.py`).

If you create a function in the console using a different file name or function handler name, you must edit the default handler name.

To change the function handler name (console)

1. Open the Functions page of the Lambda console and choose your function.
2. Choose the Code tab.
3. Scroll down to the Runtime settings pane and choose Edit.
4. In Handler, enter the new name for your function handler.
5. Choose Save.
How it works

When Lambda invokes your function handler, the Lambda runtime passes two arguments to the function handler:

- The first argument is the event object. An event is a JSON-formatted document that contains data for a Lambda function to process. The Lambda runtime converts the event to an object and passes it to your function code. It is usually of the Python dict type. It can also be list, str, int, float, or the NoneType type.

  The event object contains information from the invoking service. When you invoke a function, you determine the structure and contents of the event. When an AWS service invokes your function, the service defines the event structure. For more information about events from AWS services, see Using AWS Lambda with other services.

- The second argument is the context object. A context object is passed to your function by Lambda at runtime. This object provides methods and properties that provide information about the invocation, function, and runtime environment.

Returning a value

Optionally, a handler can return a value. What happens to the returned value depends on the invocation type and the service that invoked the function. For example:

- If you use the RequestResponse invocation type, such as Synchronous invocation, AWS Lambda returns the result of the Python function call to the client invoking the Lambda function (in the HTTP response to the invocation request, serialized into JSON). For example, AWS Lambda console uses the RequestResponse invocation type, so when you invoke the function on the console, the console will display the returned value.

- If the handler returns objects that can't be serialized by json.dumps, the runtime returns an error.

- If the handler returns None, as Python functions without a return statement implicitly do, the runtime returns null.

- If you use the Event invocation type (an asynchronous invocation), the value is discarded.
Note

In Python 3.9 and later releases, Lambda includes the requestId of the invocation in the error response.

Examples

The following section shows examples of Python functions you can use with Lambda. If you use the Lambda console to author your function, you do not need to attach a zip archive file to run the functions in this section. These functions use standard Python libraries which are included with the Lambda runtime you selected. For more information, see Lambda deployment packages.

Returning a message

The following example shows a function called lambda_handler. The function accepts user input of a first and last name, and returns a message that contains data from the event it received as input.

```python
def lambda_handler(event, context):
    message = 'Hello {} {}!'.format(event['first_name'], event['last_name'])
    return {
        'message' : message
    }
```

You can use the following event data to invoke the function:

```json
{
    "first_name": "John",
    "last_name": "Smith"
}
```

The response shows the event data passed as input:

```json
{
    "message": "Hello John Smith!"
}
```
Parsing a response

The following example shows a function called `lambda_handler`. The function uses event data passed by Lambda at runtime. It parses the environment variable in AWS_REGION returned in the JSON response.

```python
import os
import json

def lambda_handler(event, context):
    json_region = os.environ['AWS_REGION']
    return {
        "statusCode": 200,
        "headers": {
            "Content-Type": "application/json"
        },
        "body": json.dumps({
            "Region ": json_region
        })
    }
```

You can use any event data to invoke the function:

```json
{
    "key1": "value1",
    "key2": "value2",
    "key3": "value3"
}
```

Lambda runtimes set several environment variables during initialization. For more information on the environment variables returned in the response at runtime, see Using Lambda environment variables.

The function in this example depends on a successful response (in 200) from the Invoke API. For more information on the Invoke API status, see the Invoke Response Syntax.

Returning a calculation

The following example shows a function called `lambda_handler`. The function accepts user input and returns a calculation to the user. For more information about this example, see the aws-sdk-examples GitHub repository.
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    ...
    result = None
    action = event.get('action')
    if action == 'increment':
        result = event.get('number', 0) + 1
        logger.info('Calculated result of %s', result)
    else:
        logger.error('%s is not a valid action.', action)

    response = {'result': result}
    return response

You can use the following event data to invoke the function:

```json
{
    "action": "increment",
    "number": 3
}
```
Working with .zip file archives for Python Lambda functions

Your AWS Lambda function’s code comprises a .py file containing your function’s handler code, together with any additional packages and modules your code depends on. To deploy this function code to Lambda, you use a deployment package. This package may either be a .zip file archive or a container image. For more information about using container images with Python, see Deploy Python Lambda functions with container images.

To create your deployment package as .zip file archive, you can use your command-line tool’s built-in .zip file archive utility, or any other .zip file utility such as 7zip. The examples shown in the following sections assume you’re using a command-line zip tool in a Linux or MacOS environment. To use the same commands in Windows, you can install the Windows Subsystem for Linux to get a Windows-integrated version of Ubuntu and Bash.

Note that Lambda uses POSIX file permissions, so you may need to set permissions for the deployment package folder before you create the .zip file archive.

Topics

- Runtime dependencies in Python
- Creating a .zip deployment package with no dependencies
- Creating a .zip deployment package with dependencies
- Creating a Python layer for your dependencies
- Dependency search path and runtime-included libraries
- Using __pycache__ folders
- Creating .zip deployment packages with native libraries
- Creating and updating Python Lambda functions using .zip files

Runtime dependencies in Python

For Lambda functions that use the Python runtime, a dependency can be any Python package or module. When you deploy your function using a .zip archive, you can either add these dependencies to your .zip file with your function code or use a Lambda layer. A layer is a separate .zip file that can contain additional code and other content. To learn more about using Lambda layers, see Lambda layers.
The Lambda Python runtimes includes the AWS SDK for Python (Boto3) and its dependencies. Lambda provides the SDK in the runtime for deployment scenarios where you are unable to add your own dependencies. These scenarios include creating functions in the console using the built-in code editor or using inline functions in AWS Serverless Application Model (AWS SAM) or AWS CloudFormation templates.

Lambda periodically updates the libraries in the Python runtime to include the latest updates and security patches. If your function uses the version of the Boto3 SDK included in the runtime but your deployment package includes SDK dependencies, this can cause version misalignment issues. For example, your deployment package could include the SDK dependency urllib3. When Lambda updates the SDK in the runtime, compatibility issues between the new version of the runtime and the version of urllib3 in your deployment package can cause your function to fail.

⚠️ **Important**

To maintain full control over your dependencies and to avoid possible version misalignment issues, we recommend you add all of your function’s dependencies to your deployment package, even if versions of them are included in the Lambda runtime. This includes the Boto3 SDK.

Under the [AWS shared responsibility model](https://aws.amazon.com/sharedresponsibility/), you are responsible for the management of any dependencies in your functions' deployment packages. This includes applying updates and security patches. To update dependencies in your function's deployment package, first create a new .zip file and then upload it to Lambda. See [Creating a .zip deployment package with dependencies](https://docs.aws.amazon.com/lambda/latest/dg/python-deployment-package.html) and [Creating and updating Python Lambda functions using .zip files](https://docs.aws.amazon.com/lambda/latest/dg/python-deployment-package.html) for more information.

**Creating a .zip deployment package with no dependencies**

If your function code has no dependencies, your .zip file contains only the .py file with your function’s handler code. Use your preferred zip utility to create a .zip file with your .py file at the root. If the .py file is not at the root of your .zip file, Lambda won’t be able to run your code.

To learn how to deploy your .zip file to create a new Lambda function or update an existing one, see [Creating and updating Python Lambda functions using .zip files](https://docs.aws.amazon.com/lambda/latest/dg/python-deployment-package.html).
Creating a .zip deployment package with dependencies

If your function code depends on additional packages or modules, you can either add these dependencies to your .zip file with your function code or use a Lambda layer. The instructions in this section show you how to include your dependencies in your .zip deployment package. For Lambda to run your code, the .py file containing your handler code and all of your function's dependencies must be installed at the root of the .zip file.

Suppose your function code is saved in a file named `lambda_function.py`. The following example CLI commands create a .zip file named `my_deployment_package.zip` containing your function code and its dependencies. You can either install your dependencies directly to a folder in your project directory or use a Python virtual environment.

To create the deployment package (project directory)

1. Navigate to the project directory containing your `lambda_function.py` source code file. In this example, the directory is named `my_function`.
   
   ```bash
   cd my_function
   ```

2. Create a new directory named `package` into which you will install your dependencies.

   ```bash
   mkdir package
   ```

   Note that for a .zip deployment package, Lambda expects your source code and its dependencies all to be at the root of the .zip file. However, installing dependencies directly in your project directory can introduce a large number of new files and folders and make navigating around your IDE difficult. You create a separate `package` directory here to keep your dependencies separate from your source code.

3. Install your dependencies in the `package` directory. The example below installs the Boto3 SDK from the Python Package Index using pip. If your function code uses Python packages you have created yourself, save them in the `package` directory.

   ```bash
   pip install --target ./package boto3
   ```

4. Create a .zip file with the installed libraries at the root.

   ```bash
   cd package
   ```
This generates a `my_deployment_package.zip` file in your project directory.

5. Add the `lambda_function.py` file to the root of the `.zip` file

```bash
cd ..
zip my_deployment_package.zip lambda_function.py
```

Your `.zip` file should have a flat directory structure, with your function's handler code and all your dependency folders installed at the root as follows.

```
my_deployment_package.zip
|-- bin
  |-- jp.py
|-- boto3
  |-- compat.py
  |-- data
  |-- docs
  ...  
|-- lambda_function.py
```

If the `.py` file containing your function's handler code is not at the root of your `.zip` file, Lambda will not be able to run your code.

**To create the deployment package (virtual environment)**

1. Create and activate a virtual environment in your project directory. In this example the project directory is named `my_function`.

```bash
~$ cd my_function
~/my_function$ python3.12 -m venv my_virtual_env
~/my_function$ source ./my_virtual_env/bin/activate
```

2. Install your required libraries using `pip`. The following example installs the Boto3 SDK

```bash
(my_virtual_env) ~/my_function$ pip install boto3
```

3. Use `pip show` to find the location in your virtual environment where `pip` has installed your dependencies.
The folder in which pip installs your libraries may be named site-packages or dist-packages. This folder may be located in either the lib/python3.x or lib64/python3.x directory (where python3.x represents the version of Python you are using).

4. Deactivate the virtual environment

   (my_virtual_env) ~/my_function$ deactivate

5. Navigate into the directory containing the dependencies you installed with pip and create a .zip file in your project directory with the installed dependencies at the root. In this example, pip has installed your dependencies in the my_virtual_env/lib/python3.12/site-packages directory.

   ~/my_function$ cd my_virtual_env/lib/python3.12/site-packages

6. Navigate to the root of your project directory where the .py file containing your handler code is located and add that file to the root of your .zip package. In this example, your function code file is named lambda_function.py.

   ~/my_function/my_virtual_env/lib/python3.12/site-packages$ cd ..../..../..../
   ~/my_function$ zip my_deployment_package.zip lambda_function.py

**Creating a Python layer for your dependencies**

The instructions in this section show you how to include your dependencies in a layer. For instructions on how to include your dependencies in your deployment package, see the section called “Creating a .zip deployment package with dependencies”.

When you add a layer to a function, Lambda loads the layer content into the /opt directory of that execution environment. For each Lambda runtime, the PATH variable already includes specific folder paths within the /opt directory. To ensure that your layer content gets picked up by the PATH variable, include the content in the following folder paths:

- python
• python/lib/python3.10/site-packages (site directories)

For example, your layer .zip file structure might look like the following:

```
pillow.zip
# python/PIL
# python/Pillow-5.3.0.dist-info
```

In addition, Lambda automatically detects any libraries in the /opt/lib directory, and any binaries in the /opt/bin directory. To ensure that Lambda properly finds your layer content, you can also create a layer with the following structure:

```
custom-layer.zip
# lib
  | lib_1
  | lib_2
# bin
  | bin_1
  | bin_2
```

After you package your layer, see the section called “Creating and deleting layers” and the section called “Adding layers” to complete your layer setup.

**Dependency search path and runtime-included libraries**

When you use an import statement in your code, the Python runtime searches the directories in its search path until it finds the module or package. By default, the first location the runtime searches is the directory into which your .zip deployment package is decompressed and mounted (/var/task). If you include a version of a runtime-included library in your deployment package, your version will take precedence over the version that's included in the runtime. Dependencies in your deployment package also have precedence over dependencies in layers.

When you add a dependency to a layer, Lambda extracts this to /opt/python/lib/python3.x/site-packages (where python3.x represents the version of the runtime you're using) or /opt/python. In the search path, these directories have precedence over the directories containing the runtime-included libraries and pip-installed libraries (/var/runtime and /var/lang/lib/python3.x/site-packages). Libraries in function layers therefore have precedence over versions included in the runtime.
In the Python 3.11 managed runtime and base image, the AWS SDK and its dependencies are installed in the `/var/lang/lib/python3.11/site-packages` directory.

You can see the full search path for your Lambda function by adding the following code snippet.

```python
import sys
search_path = sys.path
print(search_path)
```

Because dependencies in your deployment package or layers take precedence over runtime-included libraries, this can cause version misalignment problems if you include an SDK dependency such as `urllib3` in your package without including the SDK as well. If you deploy your own version of a Boto3 dependency, you must also deploy Boto3 as a dependency in your deployment package. We recommend that you package all of your function’s dependencies, even if versions of them are included in the runtime.

You can also add dependencies in a separate folder inside your `.zip` package. For example, you might add a version of the Boto3 SDK to a folder in your `.zip` package called `common`. When your `.zip` package is decompressed and mounted, this folder is placed inside the `/var/task` directory. To use a dependency from a folder in your `.zip` deployment package in your code, use an `import from` statement. For example, to use a version of Boto3 from a folder named `common` in your `.zip` package, use the following statement.

```python
from common import boto3
```

**Using __pycache__ folders**

We recommend that you don’t include `__pycache__` folders in your function's deployment package. Python bytecode that's compiled on a build machine with a different architecture or operating system might not be compatible with the Lambda execution environment.
Creating .zip deployment packages with native libraries

If your function uses only pure Python packages and modules, you can use the `pip install` command to install your dependencies on any local build machine and create your .zip file. Many popular Python libraries, including NumPy and Pandas, are not pure Python and contain code written in C or C++. When you add libraries containing C/C++ code to your deployment package, you must build your package correctly to ensure that it’s compatible with the Lambda execution environment.

Most packages available on the Python Package Index (PyPI) are available as “wheels” (.whl files). A .whl file is a type of ZIP file which contains a built distribution with pre-compiled binaries for a particular operating system and instruction set architecture. To make your deployment package compatible with Lambda, you install the wheel for Linux operating systems and your function’s instruction set architecture.

Some packages may only be available as source distributions. For these packages, you need to compile and build the C/C++ components yourself.

To see what distributions are available for your required package, do the following:

1. Search for the name of the package on the Python Package Index main page.
2. Choose the version of the package you want to use.
3. Choose Download files.

Working with built distributions (wheels)

To download a wheel that's compatible with Lambda, you use the `pip --platform` option.

If your Lambda function uses the x86_64 instruction set architecture, run the following `pip install` command to install a compatible wheel in your package directory. Replace `--python 3.x` with the version of the Python runtime you are using.

```
pip install \
  --platform manylinux2014_x86_64 \
  --target=package \
  --implementation cp \
  --python-version 3.x \
  --only-binary=:all: --upgrade \
  <package_name>
```
If your function uses the **arm64** instruction set architecture, run the following command. Replace `--python 3.x` with the version of the Python runtime you are using.

```
pip install \
--platform manylinux2014_aarch64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

**Working with source distributions**

If your package is only available as a source distribution, you need to build the C/C++ libraries yourself. To make your package compatible with the Lambda execution environment, you need to build it in an environment that uses the same Amazon Linux 2 operating system. You can do this by building your package in an Amazon EC2 Linux instance.

To learn how to launch and connect to an Amazon EC2 Linux instance, see [Tutorial: Get started with Amazon EC2 Linux instances](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/quick[start]_tutorial.html) in the *Amazon EC2 User Guide for Linux Instances*.

**Creating and updating Python Lambda functions using .zip files**

Once you have created your .zip deployment package, you can use it to create a new Lambda function or update an existing one. You can deploy your .zip package using the Lambda console, the AWS Command Line Interface, and the Lambda API. You can also create and update Lambda functions using AWS Serverless Application Model (AWS SAM) and AWS CloudFormation.

The maximum size for a .zip deployment package for Lambda is 250 MB (unzipped). Note that this limit applies to the combined size of all the files you upload, including any Lambda layers.

**Creating and updating functions with .zip files using the console**

To create a new function, you must first create the function in the console, then upload your .zip archive. To update an existing function, open the page for your function, then follow the same procedure to add your updated .zip file.

If your .zip file is less than 50MB, you can create or update a function by uploading the file directly from your local machine. For .zip files greater than 50MB, you must upload your package to an Amazon S3 bucket first. For instructions on how to upload a file to an Amazon S3 bucket using the
AWS Management Console, see Getting started with Amazon S3. To upload files using the AWS CLI, see Move objects in the AWS CLI User Guide.

Note
You cannot convert an existing container image function to use a .zip archive. You must create a new function.

To create a new function (console)

1. Open the Functions page of the Lambda console and choose Create Function.
2. Choose Author from scratch.
3. Under Basic information, do the following:
   a. For Function name, enter the name for your function.
   b. For Runtime, select the runtime you want to use.
   c. (Optional) For Architecture, choose the instruction set architecture for your function. The default architecture is x86_64. Ensure that the .zip deployment package for your function is compatible with the instruction set architecture you select.
4. (Optional) Under Permissions, expand Change default execution role. You can create a new Execution role or use an existing one.
5. Choose Create function. Lambda creates a basic 'Hello world' function using your chosen runtime.

To upload a .zip archive from your local machine (console)

1. In the Functions page of the Lambda console, choose the function you want to upload the .zip file for.
2. Select the Code tab.
3. In the Code source pane, choose Upload from.
5. To upload the .zip file, do the following:
   a. Select Upload, then select your .zip file in the file chooser.
   b. Choose Open.
c. Choose **Save**.

**To upload a .zip archive from an Amazon S3 bucket (console)**

1. In the [Functions page](#) of the Lambda console, choose the function you want to upload a new .zip file for.
2. Select the **Code** tab.
3. In the **Code source** pane, choose **Upload from**.
4. Choose **Amazon S3 location**.
5. Paste the Amazon S3 link URL of your .zip file and choose **Save**.

**Updating .zip file functions using the console code editor**

For some functions with .zip deployment packages, you can use the Lambda console's built-in code editor to update your function code directly. To use this feature, your function must meet the following criteria:

- Your function must use one of the interpreted language runtimes (Python, Node.js, or Ruby)
- Your function’s deployment package must be smaller than 3MB.

Function code for functions with container image deployment packages cannot be edited directly in the console.

**To update function code using the console code editor**

1. Open the [Functions page](#) of the Lambda console and select your function.
2. Select the **Code** tab.
3. In the **Code source** pane, select your source code file and edit it in the integrated code editor.
4. When you have finished editing your code, choose **Deploy** to save your changes and update your function.

**Creating and updating functions with .zip files using the AWS CLI**

You can use the [AWS CLI](#) to create a new function or to update an existing one using a .zip file. Use the `create-function` and `update-function-code` commands to deploy your .zip package.
If your .zip file is smaller than 50MB, you can upload the .zip package from a file location on your local build machine. For larger files, you must upload your .zip package from an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see [Move objects](#) in the *AWS CLI User Guide*.

### Note

If you upload your .zip file from an Amazon S3 bucket using the AWS CLI, the bucket must be located in the same AWS Region as your function.

To create a new function using a .zip file with the AWS CLI, you must specify the following:

- The name of your function (`--function-name`)
- Your function's runtime (`--runtime`)
- The Amazon Resource Name (ARN) of your function's execution role (`--role`)
- The name of the handler method in your function code (`--handler`)

You must also specify the location of your .zip file. If your .zip file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.

```
aws lambda create-function --function-name myFunction \
--runtime python3.12 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the `--code` option as shown in the following example command. You only need to use the `S3ObjectVersion` parameter for versioned objects.

```
aws lambda create-function --function-name myFunction \
--runtime python3.12 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=myBucketName,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

To update an existing function using the CLI, you specify the the name of your function using the `--function-name` parameter. You must also specify the location of the .zip file you want to use...
to update your function code. If your .zip file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the `--s3-bucket` and `--s3-key` options as shown in the following example command. You only need to use the `--s3-object-version` parameter for versioned objects.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket myBucketName --s3-key myFileName.zip --s3-object-version myObject Version
```

**Creating and updating functions with .zip files using the Lambda API**

To create and update functions using a .zip file archive, use the following API operations:

- [CreateFunction](#)
- [UpdateFunctionCode](#)

**Creating and updating functions with .zip files using AWS SAM**

The AWS Serverless Application Model (AWS SAM) is a toolkit that helps streamline the process of building and running serverless applications on AWS. You define the resources for your application in a YAML or JSON template and use the AWS SAM command line interface (AWS SAM CLI) to build, package, and deploy your applications. When you build a Lambda function from an AWS SAM template, AWS SAM automatically creates a .zip deployment package or container image with your function code and any dependencies you specify. To learn more about using AWS SAM to build and deploy Lambda functions, see [Getting started with AWS SAM](#) in the [AWS Serverless Application Model Developer Guide](#).

You can also use AWS SAM to create a Lambda function using an existing .zip file archive. To create a Lambda function using AWS SAM, you can save your .zip file in an Amazon S3 bucket or in a local folder on your build machine. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see [Move objects](#) in the [AWS CLI User Guide](#).

In your AWS SAM template, the `AWS::Serverless::Function` resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:
• PackageType - set to Zip
• CodeUri - set to the function code's Amazon S3 URI, path to local folder, or FunctionCode object
• Runtime - Set to your chosen runtime

With AWS SAM, if your .zip file is larger than 50MB, you don't need to upload it to an Amazon S3 bucket first. AWS SAM can upload .zip packages up to the maximum allowed size of 250MB (unzipped) from a location on your local build machine.

To learn more about deploying functions using .zip file in AWS SAM, see AWS::Serverless::Function in the AWS SAM Developer Guide.

Creating and updating functions with .zip files using AWS CloudFormation

You can use AWS CloudFormation to create a Lambda function using a .zip file archive. To create a Lambda function from a .zip file, you must first upload your file to an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the AWS CLI User Guide.

For Node.js and Python runtimes, you can also provide inline source code in your AWS CloudFormation template. AWS CloudFormation then creates a .zip file containing your code when you build your function.

Using an existing .zip file

In your AWS CloudFormation template, the AWS::Lambda::Function resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

• PackageType - Set to Zip
• Code - Enter the Amazon S3 bucket name and the .zip file name in the S3Bucket and S3Key fields
• Runtime - Set to your chosen runtime

Creating a .zip file from inline code

You can declare simple functions written in Python or Node.js inline in an AWS CloudFormation template. Because the code is embedded in YAML or JSON, you can't add any external dependencies to your deployment package. This means your function has to use the version of the AWS SDK that's included in the runtime. The requirements of the template, such as having
to escape certain characters, also make it harder to use your IDE's syntax checking and code completion features. This means that your template might require additional testing. Because of these limitations, declaring functions inline is best suited for very simple code that does not change frequently.

To create a .zip file from inline code for Node.js and Python runtimes, set the following properties in your template's `AWS::Lambda::Function` resource:

- **PackageType** - Set to *Zip*
- **Code** - Enter your function code in the *ZipFile* field
- **Runtime** - Set to your chosen runtime

The .zip file that AWS CloudFormation generates cannot exceed 4MB. To learn more about deploying functions using .zip file in AWS CloudFormation, see `AWS::Lambda::Function` in the *AWS CloudFormation User Guide*. 

Creating and updating Python Lambda functions using .zip files
Deploy Python Lambda functions with container images

There are three ways to build a container image for a Python Lambda function:

- **Using an AWS base image for Python**

  The [AWS base images](https://docs.aws.amazon.com/lambda/latest/dg/lambda-container-images.html) are preloaded with a language runtime, a runtime interface client to manage the interaction between Lambda and your function code, and a runtime interface emulator for local testing.

- **Using an AWS OS-only base image**

  [AWS OS-only base images](https://docs.aws.amazon.com/lambda/latest/dg/lambda-container-images.html) contain an Amazon Linux distribution and the runtime interface emulator. These images are commonly used to create container images for compiled languages, such as Go and Rust, and for a language or language version that Lambda doesn't provide a base image for, such as Node.js 19. You can also use OS-only base images to implement a custom runtime. To make the image compatible with Lambda, you must include the runtime interface client for Python in the image.

- **Using a non-AWS base image**

  You can use an alternative base image from another container registry, such as Alpine Linux or Debian. You can also use a custom image created by your organization. To make the image compatible with Lambda, you must include the runtime interface client for Python in the image.

### Tip

To reduce the time it takes for Lambda container functions to become active, see [Use multi-stage builds](https://docs.aws.amazon.com/lambda/latest/dg/lambda-container-images.html) in the Docker documentation. To build efficient container images, follow the [Best practices for writing Dockerfiles](https://docs.aws.amazon.com/lambda/latest/dg/lambda-container-images.html).

This page explains how to build, test, and deploy container images for Lambda.

**Topics**

- AWS base images for Python
- Using an AWS base image for Python
- Using an alternative base image with the runtime interface client
AWS base images for Python

AWS provides the following base images for Python:

<table>
<thead>
<tr>
<th>Tags</th>
<th>Runtime</th>
<th>Operating system</th>
<th>Dockerfile</th>
<th>Deprecation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.12</td>
<td>Python 3.12</td>
<td>Amazon Linux 2023</td>
<td>[Dockerfile for Python 3.12 on GitHub]</td>
<td></td>
</tr>
<tr>
<td>3.11</td>
<td>Python 3.11</td>
<td>Amazon Linux 2</td>
<td>[Dockerfile for Python 3.11 on GitHub]</td>
<td></td>
</tr>
<tr>
<td>3.10</td>
<td>Python 3.10</td>
<td>Amazon Linux 2</td>
<td>[Dockerfile for Python 3.10 on GitHub]</td>
<td></td>
</tr>
<tr>
<td>3.9</td>
<td>Python 3.9</td>
<td>Amazon Linux 2</td>
<td>[Dockerfile for Python 3.9 on GitHub]</td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Python 3.8</td>
<td>Amazon Linux 2</td>
<td>[Dockerfile for Python 3.8 on GitHub]</td>
<td>Oct 14, 2024</td>
</tr>
</tbody>
</table>

Note

When upgrading to the Python 3.12 base image, you must update your Dockerfiles to use `dnf` as the package manager instead of `yum`. For more information, see [Building Lambda functions with Python](https://docs.aws.amazon.com/lambda/latest/dg/python.html).

Amazon ECR repository: [gallery.ecr.aws/lambda/python](https://ecr.amazonaws.com/gallery.ecr.aws/lambda/python)

Dependency search path in the base images

When you use an `import` statement in your code, the Python runtime searches the directories in its search path until it finds the module or package. By default, the runtime searches the `{LAMBDA_TASK_ROOT}` directory first. If you include a version of a runtime-included library in your image, your version will take precedence over the version that's included in the runtime.
Other steps in the search path depend on which version of the Lambda base image for Python you're using:

- **Python 3.11 and later**: Runtime-included libraries and pip-installed libraries are installed in the `/var/lang/lib/python3.11/site-packages` directory. This directory has precedence over `/var/runtime` in the search path. You can override the SDK by using pip to install a newer version. You can use pip to verify that the runtime-included SDK and its dependencies are compatible with any packages that you install.

- **Python 3.8-3.10**: Runtime-included libraries are installed in the `/var/runtime` directory. Pip-installed libraries are installed in the `/var/lang/lib/python3.x/site-packages` directory. The `/var/runtime` directory has precedence over `/var/lang/lib/python3.x/site-packages` in the search path.

You can see the full search path for your Lambda function by adding the following code snippet.

```python
import sys

search_path = sys.path
print(search_path)
```

### Using an AWS base image for Python

#### Prerequisites

To complete the steps in this section, you must have the following:

- [AWS Command Line Interface (AWS CLI) version 2](#)
- [Docker](#)
- Python

#### Creating an image from a base image

**To create a container image from an AWS base image for Python**

1. Create a directory for the project, and then switch to that directory.

```bash
mkdir example
cd example
```
2. Create a new file called `lambda_function.py`. You can add the following sample function code to the file for testing, or use your own.

   **Example Python function**
   ```python
   import sys
   def handler(event, context):
       return 'Hello from AWS Lambda using Python' + sys.version + '!
   ```

3. Create a new file called `requirements.txt`. If you’re using the sample function code from the previous step, you can leave the file empty because there are no dependencies. Otherwise, list each required library. For example, here’s what your `requirements.txt` should look like if your function uses the AWS SDK for Python (Boto3):

   **Example requirements.txt**
   ```
   boto3
   ```

4. Create a new Dockerfile with the following configuration:
   - Set the `FROM` property to the [URI of the base image](#).
   - Use the `COPY` command to copy the function code and runtime dependencies to `{LAMBDA_TASK_ROOT}`.
   - Set the `CMD` argument to the Lambda function handler.

   **Example Dockerfile**
   ```
   FROM public.ecr.aws/lambda/python:3.12
   
   # Copy requirements.txt
   COPY requirements.txt ${LAMBDA_TASK_ROOT}
   
   # Install the specified packages
   RUN pip install -r requirements.txt
   
   # Copy function code
   COPY lambda_function.py ${LAMBDA_TASK_ROOT}
   
   # Set the CMD to your handler (could also be done as a parameter override outside of the Dockerfile)
   ```
5. Build the Docker image with the `docker build` command. The following example names the image `docker-image` and gives it the test tag.

```bash
docker build --platform linux/amd64 -t docker-image:test .
```

**Note**

The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

1. Start the Docker image with the `docker run` command. In this example, `docker-image` is the image name and `test` is the tag.

```bash
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

**Note**

If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

2. From a new terminal window, post an event to the local endpoint.

**Linux/macOS**

In Linux and macOS, run the following `curl` command:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```
This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{"payload":"hello world!"}'
```

PowerShell

In PowerShell, run the following Invoke-WebRequest command:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{}' -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType "application/json"
```

3. Get the container ID.

```bash
docker ps
```

4. Use the `docker kill` command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```bash
docker kill 3766c4ab331c
```

**Deploying the image**

To upload the image to Amazon ECR and create the Lambda function

1. Run the `get-login-password` command to authenticate the Docker CLI to your Amazon ECR registry.
• Set the --region value to the AWS Region where you want to create the Amazon ECR repository.

• Replace 111122223333 with your AWS account ID.

```bash
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a repository in Amazon ECR using the `create-repository` command.

```bash
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

Note

The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```json
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copy the `repositoryUri` from the output in the previous step.
4. Run the `docker tag` command to tag your local image into your Amazon ECR repository as the latest version. In this command:

- Replace `docker-image:test` with the name and `tag` of your Docker image.
- Replace `<ECRrepositoryUri>` with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Example:
```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the `docker push` command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.
```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. **Create an execution role** for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.
```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

**Note**

You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see [Amazon ECR cross-account permissions](https://docs.aws.amazon.com/AmazonECR/latest/userguide/cross-account-permissions.html).

8. Invoke the function.
aws lambda invoke --function-name hello-world response.json

You should see a response like this:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the `update-function-code` command to deploy the image to the Lambda function.

**Using an alternative base image with the runtime interface client**

If you use an OS-only base image or an alternative base image, you must include the runtime interface client in your image. The runtime interface client extends the Lambda runtime API, which manages the interaction between Lambda and your function code.

Install the the runtime interface client for Python using the pip package manager:

```
pip install awslambdaric
```

You can also download the Python runtime interface client from GitHub.

The following example demonstrates how to build a container image for Python using a non-AWS base image. The example Dockerfile uses an official Python base image. The Dockerfile includes the runtime interface client for Python.

**Prerequisites**

To complete the steps in this section, you must have the following:

- AWS Command Line Interface (AWS CLI) version 2
- Docker
- Python
Creating an image from an alternative base image

To create a container image from a non-AWS base image

1. Create a directory for the project, and then switch to that directory.

   mkdir example
   cd example

2. Create a new file called `lambda_function.py`. You can add the following sample function code to the file for testing, or use your own.

   **Example Python function**

   ```python
   import sys
   def handler(event, context):
       return 'Hello from AWS Lambda using Python' + sys.version + '!'  
   ```

3. Create a new file called `requirements.txt`. If you're using the sample function code from the previous step, you can leave the file empty because there are no dependencies. Otherwise, list each required library. For example, here's what your `requirements.txt` should look like if your function uses the AWS SDK for Python (Boto3):

   **Example requirements.txt**

   ```
boto3
   ```

4. Create a new Dockerfile. The following Dockerfile uses an official Python base image instead of an AWS base image. The Dockerfile includes the runtime interface client, which makes the image compatible with Lambda. The following example Dockerfile uses a multi-stage build.

   - Set the `FROM` property to the base image.
   - Set the `ENTRYPOINT` to the module that you want the Docker container to run when it starts. In this case, the module is the runtime interface client.
   - Set the `CMD` to the Lambda function handler.

   **Example Dockerfile**

   ```
   # Define custom function directory
   ARG FUNCTION_DIR="/function"
   ```
FROM `python:3.12` as build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}

# Install the function's dependencies
RUN pip install --target ${FUNCTION_DIR} awslambdaric

# Use a slim version of the base Python image to reduce the final image size
FROM `python:3.12-slim`

# Include global arg in this stage of the build
ARG FUNCTION_DIR
# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/local/bin/python", "-m", "awslambdaric" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "lambda_function.handler" ]

5. Build the Docker image with the `docker build` command. The following example names the image `docker-image` and gives it the `test` tag.

docker build --platform linux/amd64 -t docker-image:test .

**Note**

The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using...
the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

Use the runtime interface emulator to locally test the image. You can build the emulator into your image or install it on your local machine.

To install and run the runtime interface emulator on your local machine

1. From your project directory, run the following command to download the runtime interface emulator (x86-64 architecture) from GitHub and install it on your local machine.

   **Linux/macOS**

   ```bash
   mkdir -p ~/.aws-lambda-rie && \
   chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   To install the arm64 emulator, replace the GitHub repository URL in the previous command with the following:

   ```bash
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   **PowerShell**

   ```powershell
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

   To install the arm64 emulator, replace the `$downloadLink` with the following:
2. Start the Docker image with the `docker run` command. Note the following:

- `docker-image` is the image name and `test` is the tag.
- `/usr/local/bin/python -m awslambdaric lambda_function.handler` is the `ENTRYPOINT` followed by the `CMD` from your Dockerfile.

**Linux/macOS**

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p 9000:8080 \
  --entrypoint /aws-lambda/aws-lambda-rie \
  docker-image:test \
  /usr/local/bin/python -m awslambdaric lambda_function.handler
```

**PowerShell**

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p 9000:8080 `\`
  --entrypoint /aws-lambda/aws-lambda-rie `\`
  docker-image:test `\`
  /usr/local/bin/python -m awslambdaric lambda_function.handler
```

This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

**Note**

If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

3. Post an event to the local endpoint.

**Linux/macOS**

In Linux and macOS, run the following `curl` command:
This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

**PowerShell**

In PowerShell, run the following `Invoke-WebRequest` command:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{}' -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType "application/json"
```

4. Get the container ID.

```bash
docker ps
```

5. Use the `docker kill` command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```bash
docker kill 3766c4ab331c
```
Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the `get-login-password` command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace 111122223333 with your AWS account ID.

   ```
   aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
   ```

2. Create a repository in Amazon ECR using the `create-repository` command.

   ```
   aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
   ```

   **Note**
   The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```json
{
   "repository": {
      "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
      "registryId": "111122223333",
      "repositoryName": "hello-world",
      "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
      "createdAt": "2023-03-09T10:39:01+00:00",
      "imageTagMutability": "MUTABLE",
      "imageScanningConfiguration": {
         "scanOnPush": true
      },
      "encryptionConfiguration": {

   ```
```
3. Copy the repositoryUri from the output in the previous step.

4. Run the `docker tag` command to tag your local image into your Amazon ECR repository as the latest version. In this command:
   - Replace `docker-image:test` with the name and `tag` of your Docker image.
   - Replace `<ECRrepositoryUri>` with the repositoryUri that you copied. Make sure to include :latest at the end of the URI.

   ```bash
docker tag docker-image:test <ECRrepositoryUri>:latest
```

   Example:

   ```bash
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the `docker push` command to deploy your local image to the Amazon ECR repository. Make sure to include :latest at the end of the repository URI.

   ```bash
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include :latest at the end of the URI.

   ```bash
   aws lambda create-function \
   --function-name hello-world \
   --package-type Image \
   --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
   --role arn:aws:iam::111122223333:role/lambda-ex
   ```
Note

You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```bash
```

You should see a response like this:

```json
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the `update-function-code` command to deploy the image to the Lambda function.

For an example of how to create a Python image from an Alpine base image, see Container image support for Lambda on the AWS Blog.
AWS Lambda context object in Python

When Lambda runs your function, it passes a context object to the handler. This object provides methods and properties that provide information about the invocation, function, and execution environment. For more information on how the context object is passed to the function handler, see Lambda function handler in Python.

Context methods

- `get_remaining_time_in_millis` – Returns the number of milliseconds left before the execution times out.

Context properties

- `function_name` – The name of the Lambda function.
- `function_version` – The version of the function.
- `invoked_function_arn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memory_limit_in_meb` – The amount of memory that's allocated for the function.
- `aws_request_id` – The identifier of the invocation request.
- `log_group_name` – The log group for the function.
- `log_stream_name` – The log stream for the function instance.
- `identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
  - `cognito_identity_id` – The authenticated Amazon Cognito identity.
  - `cognito_identity_pool_id` – The Amazon Cognito identity pool that authorized the invocation.
- `client_context` – (mobile apps) Client context that's provided to Lambda by the client application.
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
• **custom** – A dict of custom values set by the mobile client application.
• **env** – A dict of environment information provided by the AWS SDK.

The following example shows a handler function that logs context information.

**Example handler.py**

```python
import time

def lambda_handler(event, context):
    print("Lambda function ARN:", context.invoked_function_arn)
    print("CloudWatch log stream name:", context.log_stream_name)
    print("CloudWatch log group name:", context.log_group_name)
    print("Lambda Request ID:", context.aws_request_id)
    print("Lambda function memory limits in MB:", context.memory_limit_in_mb)
    # We have added a 1 second delay so you can see the time remaining in
get_remaining_time_in_millis.
    time.sleep(1)
    print("Lambda time remaining in MS:", context.get_remaining_time_in_millis())
```

In addition to the options listed above, you can also use the AWS X-Ray SDK for [Instrumenting Python code in AWS Lambda](https://docs.aws.amazon.com/xray/latest/devguide/instrumentation-python.html) to identify critical code paths, trace their performance and capture the data for analysis.
AWS Lambda automatically monitors Lambda functions and sends log entries to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation and other output from your function's code to the log stream. For more information about CloudWatch Logs, see Using Amazon CloudWatch logs with AWS Lambda.

To output logs from your function code, you can use the built-in `logging` module. For more detailed entries, you can use any logging library that writes to `stdout` or `stderr`.

**Printing to the log**

To send basic output to the logs, you can use a `print` method in your function. The following example logs the values of the CloudWatch Logs log group and stream, and the event object.

Note that if your function outputs logs using Python `print` statements, Lambda can only send log outputs to CloudWatch Logs in plain text format. To capture logs in structured JSON, you need to use a supported logging library. See the section called “Using Lambda advanced logging controls with Python” for more information.

**Example lambda_function.py**

```python
import os

def lambda_handler(event, context):
    print('## ENVIRONMENT VARIABLES')
    print(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    print(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    print('## EVENT')
    print(event)
```

**Example log output**

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
/aws/lambda/my-function
2023/08/31/[$LATEST]3893xmpl17fac4485b47bb75b671a283c
## EVENT
{'key': 'value'}
```
The Python runtime logs the START, END, and REPORT lines for each invocation. The REPORT line includes the following data:

**REPORT line data fields**

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TraceId** – For traced requests, the AWS X-Ray trace ID.
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

**Using a logging library**

For more detailed logs, use the [logging](https://docs.python.org/3/library/logging.html) module in the standard library, or any third party logging library that writes to `stdout` or `stderr`.

For supported Python runtimes, you can choose whether logs created using the standard `logging` module are captured in plain text or JSON. To learn more, see the [section called “Using Lambda advanced logging controls with Python”](https://docs.aws.amazon.com/lambda/latest/dg/python-logging.html).

Currently, the default log format for all Python runtimes is plain text. The following example shows how log outputs created using the standard `logging` module are captured in plain text in CloudWatch Logs.

```python
import os
```
import logging
logger = logging.getLogger()
logger.setLevel("INFO")

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES')
    logger.info(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    logger.info(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    logger.info('## EVENT')
    logger.info(event)

The output from logger includes the log level, timestamp, and request ID.

START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO] 2023-08-31T22:12:58.534Z 1c8df7d3-xmpl-46da-9778-518e6eca8125 ##
ENVIRONMENT VARIABLES
[INFO] 2023-08-31T22:12:58.534Z 1c8df7d3-xmpl-46da-9778-518e6eca8125 /aws/
lambda/my-function
[INFO] 2023-08-31T22:12:58.534Z 1c8df7d3-xmpl-46da-9778-518e6eca8125 2023/01/31/
[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d
[INFO] 2023-08-31T22:12:58.535Z 1c8df7d3-xmpl-46da-9778-518e6eca8125 ## EVENT
[INFO] 2023-08-31T22:12:58.535Z 1c8df7d3-xmpl-46da-9778-518e6eca8125 {'key':
'value'}
END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Duration: 2.75 ms Billed
Duration: 3 ms Memory Size: 128 MB Max Memory Used: 56 MB Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370 SegmentId: 073xmpl3e442861
Sampled: true

⚠️ Note

When your function's log format is set to plain text, the default log-level setting for Python runtimes is WARN. This means that Lambda only sends log outputs of level WARN and lower to CloudWatch Logs. To change the default log level, use the Python logging setLevel() method as shown in this example code. If you set your function's log format to JSON, we recommend that you configure your function's log level using Lambda Advanced Logging Controls and not by setting the log level in code. To learn more, see the section called “Using log-level filtering with Python”
Using Lambda advanced logging controls with Python

To give you more control over how your functions' logs are captured, processed, and consumed, you can configure the following logging options for supported Lambda Python runtimes:

- **Log format** - select between plain text and structured JSON format for your function's logs
- **Log level** - for logs in JSON format, choose the detail level of the logs Lambda sends to Amazon CloudWatch, such as ERROR, DEBUG, or INFO
- **Log group** - choose the CloudWatch log group your function sends logs to

For more information about these logging options, and instructions on how to configure your function to use them, see the section called “Configuring advanced logging controls for your Lambda function”.

To learn more about using the log format and log level options with your Python Lambda functions, see the guidance in the following sections.

Using structured JSON logs with Python

If you select JSON for your function's log format, Lambda will send logs output by the Python standard logging library to CloudWatch as structured JSON. Each JSON log object contains at least four key value pairs with the following keys:

- "timestamp" - the time the log message was generated
- "level" - the log level assigned to the message
- "message" - the contents of the log message
- "requestId" - the unique request ID for the function invocation

The Python logging library can also add extra key value pairs such as "logger" to this JSON object.

The examples in the following sections show how log outputs generated using the Python logging library are captured in CloudWatch Logs when you configure your function's log format as JSON.

Note that if you use the print method to produce basic log outputs as described in the section called “Printing to the log”, Lambda will capture these outputs as plain text, even if you configure your function's logging format as JSON.
Standard JSON log outputs using Python logging library

The following example code snippet and log output show how standard log outputs generated using the Python logging library are captured in CloudWatch Logs when your function's log format is set to JSON.

Example Python logging code

```python
import logging
logger = logging.getLogger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON log record

```json
{
    "timestamp":"2023-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"Inside the handler function",
    "logger": "root",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

Logging extra parameters in JSON

When outputting JSON formatted logs using the standard Python logging library, you can also log additional parameters by using the extra keyword to pass a Python dictionary to the log output.

Example Python logging code

```python
import logging

def lambda_handler(event, context):
    logging.info(
        "extra",
        extra={"a": "b", "b": [3]},
    )
```
Example JSON log record

```json
{
  "timestamp": "2023-11-02T15:26:28Z",
  "level": "INFO",
  "message": "extra parameters example",
  "logger": "root",
  "requestId": "3dbd5759-65f6-45f8-8d7d-5bdc79a3bd01",
  "a": "b",
  "b": [3]
}
```

Logging exceptions in JSON

The following code snippet shows how Python exceptions are captured in your function's log output when you configure the log format as JSON. Note that log outputs generated using `logging.exception` are assigned the log level ERROR.

Example Python logging code

```python
import logging

def lambda_handler(event, context):
    try:
        raise Exception("exception")
    except:
        logging.exception("msg")
```

Example JSON log record

```json
{
  "timestamp": "2023-11-02T16:18:57Z",
  "level": "ERROR",
  "message": "msg",
  "logger": "root",
  "stackTrace": [
    " File "/var/task/lambda_function.py", line 15, in lambda_handler
    raise Exception("exception")"
  ],
}
```
JSON structured logs with other logging tools

If your code already uses another logging library, such as Powertools for AWS Lambda, to produce JSON structured logs, you don’t need to make any changes. AWS Lambda doesn’t double-encode any logs that are already JSON encoded. Even if you configure your function to use the JSON log format, your logging outputs appear in CloudWatch in the JSON structure you define.

The following example shows how log outputs generated using the Powertools for AWS Lambda package are captured in CloudWatch Logs. The format of this log output is the same whether your function’s logging configuration is set to JSON or TEXT. For more information about using Powertools for AWS Lambda, see the section called “Using Powertools for AWS Lambda (Python) and AWS SAM for structured logging” and the section called “Using Powertools for AWS Lambda (Python) and AWS CDK for structured logging”

Example Python logging code snippet (using Powertools for AWS Lambda)

```python
from aws_lambda_powertools import Logger

logger = Logger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON log record (using Powertools for AWS Lambda)

```json
{
    "level": "INFO",
    "location": "lambda_handler:7",
    "message": "Inside the handler function",
    "timestamp": "2023-10-31 22:38:21,010+0000",
    "service": "service_undefined",
    "xray_trace_id": "1-654181dc-65c15d6b0fecbddd1531ecb30"
}
```
Using log-level filtering with Python

By configuring log-level filtering, you can choose to send only logs of a certain logging level or lower to CloudWatch Logs. To learn how to configure log-level filtering for your function, see the section called “Log-level filtering”.

For AWS Lambda to filter your application logs according to their log level, your function must use JSON formatted logs. You can achieve this in two ways:

- Create log outputs using the standard Python logging library and configure your function to use JSON log formatting. AWS Lambda then filters your log outputs using the “level” key value pair in the JSON object described in the section called “Using structured JSON logs with Python”. To learn how to configure your function’s log format, see the section called “Configuring advanced logging controls for your Lambda function”.

- Use another logging library or method to create JSON structured logs in your code that include a “level” key value pair defining the level of the log output. For example, you can use Powertools for AWS Lambda to generate JSON structured log outputs from your code.

You can also use a print statement to output a JSON object containing a log level identifier. The following print statement produces a JSON formatted output where the log level is set to INFO. AWS Lambda will send the JSON object to CloudWatch Logs if your function’s logging level is set to INFO, DEBUG, or TRACE.

```python
print('{{"msg":"My log message", "level":"info"}}')
```

For Lambda to filter your function's logs, you must also include a "timestamp" key value pair in your JSON log output. The time must be specified in valid RFC 3339 timestamp format. If you don't supply a valid timestamp, Lambda will assign the log the level INFO and add a timestamp for you.

Viewing logs in Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded Code editor, you will find logs in the execution results. When you use the console test feature to invoke a function, you'll find Log output in the Details section.
### Viewing logs in CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

**To view logs on the CloudWatch console**

2. Choose the log group for your function (`/aws/lambda/your-function-name`).
3. Choose a log stream.

Each log stream corresponds to an instance of your function. A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see [Error processor sample application for AWS Lambda](https://github.com/aws-samples/aws-lambda-error-processor).

### Viewing logs with AWS CLI

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface (AWS CLI) version 2](https://aws.amazon.com/cli/)
- [AWS CLI – Quick configuration with `aws configure`](https://docs.aws.amazon.com/cli/latest/userguide/getting-started.html)

You can use the [AWS CLI](https://aws.amazon.com/cli/) to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

**Example retrieve a log ID**

The following example shows how to retrieve a log ID from the LogResult field for a function named `my-function`.

```bash
aws lambda invoke --function-name my-function out --log-type Tail
```
You should see the following output:

```
{
    "StatusCode": 200,
    "LogResult": "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTEzTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc21vb...", 
    "ExecutedVersion": "$LATEST"
}
```

**Example decode the logs**

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \ 
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --decode
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the AWS Command Line Interface User Guide for Version 2.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"", ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Duration: 79.67 ms   Billed
   Duration: 80 ms   Memory Size: 128 MB   Max Memory Used: 73 MB
```

The base64 utility is available on Linux, macOS, and **Ubuntu on Windows**. macOS users may need to use `base64 --D`.

**Example get-logs.sh script**

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.
Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-alias-index.html) in the *AWS Command Line Interface User Guide for Version 2*.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-name stream1 --limit 5
```

**Example macOS and Linux (only)**

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

**Example retrieve the last five log events**

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```json
{
  "StatusCode": 200,
  "ExecutedVersion": "$LATEST"
}
```

```
{
  "events": [
    {
      "timestamp": 1559763003171,
      "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST\n",
```
Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which logs are deleted automatically.

Tools and libraries

Powertools for AWS Lambda (Python) is a developer toolkit to implement Serverless best practices and increase developer velocity. The Logger utility provides a Lambda optimized logger which
includes additional information about function context across all your functions with output structured as JSON. Use this utility to do the following:

- Capture key fields from the Lambda context, cold start and structures logging output as JSON
- Log Lambda invocation events when instructed (disabled by default)
- Print all the logs only for a percentage of invocations via log sampling (disabled by default)
- Append additional keys to structured log at any point in time
- Use a custom log formatter (Bring Your Own Formatter) to output logs in a structure compatible with your organization’s Logging RFC

**Using Powertools for AWS Lambda (Python) and AWS SAM for structured logging**

Follow the steps below to download, build, and deploy a sample Hello World Python application with integrated [Powertools for Python](https://aws.amazon.com/serverless/powertools/) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a `hello world` message.

**Prerequisites**

To complete the steps in this section, you must have the following:

- Python 3.9
- [AWS CLI version 2](https://aws.amazon.com/cli/)
- [AWS SAM CLI version 1.75 or later](https://docs.aws.amazon.com/serverless-sam/latest/userguide/sam-cli-install.html). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](https://docs.aws.amazon.com/serverless-sam/latest/userguide/sam-cli-upgrade.html).

**Deploy a sample AWS SAM application**

1. Initialize the application using the Hello World Python template.

   ```bash
   sam init --app-template hello-world-powertools-python --name sam-app --package-type Zip --runtime python3.9 --no-tracing
   ```
2. Build the app.
   
   ```
   cd sam-app && sam build
   ```

3. Deploy the app.
   
   ```
   sam deploy --guided
   ```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.

   **Note**
   For **HelloWorldFunction may not have authorization defined, Is this okay?**, make sure to enter `y`.

5. Get the URL of the deployed application:
   
   ```
   aws cloudformation describe-stacks --stack-name sam-app --query
   'Stacks[0].Outputs[?OutputKey=="HelloWorldApi"].OutputValue' --output text
   ```

6. Invoke the API endpoint:
   
   ```
   curl GET <URL_FROM_PREVIOUS_STEP>
   ```

   If successful, you'll see this response:

   ```
   {"message":"hello world"}
   ```

7. To get the logs for the function, run `sam logs`. For more information, see Working with logs in the AWS Serverless Application Model Developer Guide.
   
   ```
   sam logs --stack-name sam-app
   ```

   The log output looks like this:

   ```
   2023/02/03/[$LATEST]ea9a64ec87294bf6b6c9026c05a01e04
   2023-02-03T14:59:50.371000 INIT_START Runtime Version:
   python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
   ```
Using Powertools for AWS Lambda (Python) and AWS SAM for structured logging
8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```
Managing log retention

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which CloudWatch automatically deletes the logs. To set up log retention, add the following to your AWS SAM template:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Using Powertools for AWS Lambda (Python) and AWS CDK for structured logging

Follow the steps below to download, build, and deploy a sample Hello World Python application with integrated Powertools for AWS Lambda (Python) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Python 3.9
- AWS CLI version 2
- AWS CDK version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.
Deploy a sample AWS CDK application

1. Create a project directory for your new application.

```bash
mkdir hello-world
cd hello-world
```

2. Initialize the app.

```bash
cdk init app --language python
```

3. Install the Python dependencies.

```bash
pip install -r requirements.txt
```

4. Create a directory `lambda_function` under the root folder.

```bash
mkdir lambda_function
cd lambda_function
```

5. Create a file `app.py` and add the following code to the file. This is the code for the Lambda function.

```python
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/metrics/
```
metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count, value=1)

# structured log
# See: https://docs.powertools.aws.dev/lambda-python/latest/core/logger/
logger.info("Hello world API - HTTP 200")
return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.powertools.aws.dev/lambda-python/latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
# ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)

6. Open the **hello_world** directory. You should see a file called **hello_world_stack.py**.

```python
cd ..
cd hello_world
```

7. Open **hello_world_stack.py** and add the following code to the file. This contains the **Lambda Constructor**, which creates the Lambda function, configures environment variables for Powertools and sets log retention to one week, and the **ApiGatewayv1 Constructor**, which creates the REST API.

```python
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)
```

Using Powertools for AWS Lambda (Python) and AWS CDK for structured logging
# Powertools Lambda Layer

```python
powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
    self,
    id="lambda-powertools",
    # At the moment we wrote this example, the aws_lambda_python_alpha CDK
    # constructor is in Alpha, so we use layer to make the example simpler
    # See https://docs.aws.amazon.com/cdk/api/v2/python/
    aws_cdk.aws_lambda_python_alpha/README.html
    # Check all Powertools layers versions here: https://
    # docs.powertools.aws.dev/lambda-python/latest/#lambda-layer
    layer_version_arn=f"arn:aws:lambda:
    {self.region}:017000801446:layer:AWSLambdaPowertoolsPythonV2:21"
)

function = lambda_.Function(self,
    'sample-app-lambda',
    runtime=lambda_.Runtime.PYTHON_3_9,
    layers=[powertools_layer],
    code = lambda_.Code.from_asset("./lambda_function/"),
    handler="app.lambda_handler",
    memory_size=128,
    timeout=Duration.seconds(3),
    architecture=lambda_.Architecture.X86_64,
    environment={
        "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
        "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
        "LOG_LEVEL": "INFO"
    }
)

apigw = apigwv1.RestApi(self, "PowertoolsAPI",
    deploy_options=apigwv1.StageOptions(stage_name="dev"))

hello_api = apigw.root.add_resource("hello")
    hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
        proxy=True))

CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. Deploy your application.

```bash
    cd ..
    cdk deploy
```
9. Get the URL of the deployed application:

```bash
aws cloudformation describe-stacks --stack-name HelloWorldStack --query 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. Invoke the API endpoint:

```bash
curl GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```json
{"message":"hello world"}
```

11. To get the logs for the function, run `sam logs`. For more information, see Working with logs in the AWS Serverless Application Model Developer Guide.

```
sam logs --stack-name HelloWorldStack
```

The log output looks like this:

```
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
2023-02-03T14:59:50.371000 INIT_START Runtime Version:
    python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-east-1::runtime:07a48df201798d627f2b950f03bb227aafab4a655a1d019c3296406f95937e2525
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.112000 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.114000 {
    "level": "INFO",
    "location": "hello:23",
    "message": "Hello world API - HTTP 200",
    "timestamp": "2023-02-03 14:59:51,113+0000",
    "service": "PowertoolsHelloWorld",
    "cold_start": true,
    "function_name": "sam-app-HelloWorldFunction-YBg8yfYt0c9j",
    "function_memory_size": "128",
    "function_arn": "arn:aws:lambda:us-east-1:11112223333:function:Sam-app-HelloWorldFunction-YBg8yfYt0c9j",
    "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
    "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
    "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
```
Using Powertools for AWS Lambda (Python) and AWS CDK for structured logging
12. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

cdk destroy
AWS Lambda function testing in Python

Note

See the Testing functions chapter for a complete introduction to techniques and best practices for testing serverless solutions.

Testing serverless functions uses traditional test types and techniques, but you must also consider testing serverless applications as a whole. Cloud-based tests will provide the most accurate measure of quality of both your functions and serverless applications.

A serverless application architecture includes managed services that provide critical application functionality through API calls. For this reason, your development cycle should include automated tests that verify functionality when your function and services interact.

If you do not create cloud-based tests, you could encounter issues due to differences between your local environment and the deployed environment. Your continuous integration process should run tests against a suite of resources provisioned in the cloud before promoting your code to the next deployment environment, such as QA, Staging, or Production.

Continue reading this short guide to learn about testing strategies for serverless applications, or visit the Serverless Test Samples repository to dive in with practical examples, specific to your chosen language and runtime.
For serverless testing, you will still write unit, integration and end-to-end tests.

- **Unit tests** - Tests that run against an isolated block of code. For example, verifying the business logic to calculate the delivery charge given a particular item and destination.
- **Integration tests** - Tests involving two or more components or services that interact, typically in a cloud environment. For example, verifying a function processes events from a queue.
- **End-to-end tests** - Tests that verify behavior across an entire application. For example, ensuring infrastructure is set up correctly and that events flow between services as expected to record a customer's order.

### Testing your serverless applications

You will generally use a mix of approaches to test your serverless application code, including testing in the cloud, testing with mocks, and occasionally testing with emulators.

#### Testing in the cloud

Testing in the cloud is valuable for all phases of testing, including unit tests, integration tests, and end-to-end tests. You run tests against code deployed in the cloud and interacting with cloud-based services. This approach provides the **most accurate** measure of quality of your code.

A convenient way to debug your Lambda function in the cloud is through the console with a test event. A **test event** is a JSON input to your function. If your function does not require input, the event can be an empty JSON document (`{}`). The console provides sample events for a variety of service integrations. After creating an event in the console, you can share it with your team to make testing easier and consistent.

⚠️ **Note**

Testing a function in the console is a quick way to get started, but automating your test cycles ensures application quality and development speed.

#### Testing tools

Tools and techniques exist to accelerate development feedback loops. For example, AWS SAM Accelerate and AWS CDK watch mode both decrease the time required to update cloud environments.
Moto is a Python library for mocking AWS services and resources, so that you can test your functions with little or no modification using decorators to intercept and simulate responses.

The validation feature of the Powertools for AWS Lambda (Python) provides decorators so you can validate input events and output responses from your Python functions.

For more information, read the blog post Unit Testing Lambda with Python and Mock AWS Services.

To reduce the latency involved with cloud deployment iterations, see AWS Serverless Application Model (AWS SAM) Accelerate, AWS Cloud Development Kit (AWS CDK) watch mode. These tools monitor your infrastructure and code for changes. They react to these changes by creating and deploying incremental updates automatically into your cloud environment.

Examples that use these tools are available in the Python Test Samples code repository.
AWS Lambda function errors in Python

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

This page describes how to view Lambda function invocation errors for the Python runtime using the Lambda console and the AWS CLI.

Sections

- How it works
- Using the Lambda console
- Using the AWS Command Line Interface (AWS CLI)
- Error handling in other AWS services
- Error examples
- Sample applications
- What's next?

How it works

When you invoke a Lambda function, Lambda receives the invocation request and validates the permissions in your execution role, verifies that the event document is a valid JSON document, and checks parameter values.

If the request passes validation, Lambda sends the request to a function instance. The Lambda runtime environment converts the event document into an object, and passes it to your function handler.

If Lambda encounters an error, it returns an exception type, message, and HTTP status code that indicates the cause of the error. The client or service that invoked the Lambda function can handle the error programatically, or pass it along to an end user. The correct error handling behavior depends on the type of application, the audience, and the source of the error.

The following list describes the range of status codes you can receive from Lambda.

2xx

A 2xx series error with a X-Amz-Function-Error header in the response indicates a Lambda runtime or function error. A 2xx series status code indicates that Lambda accepted the request,
but instead of an error code, Lambda indicates the error by including the X-Amz-Function-Error header in the response.

4xx

A 4xx series error indicates an error that the invoking client or service can fix by modifying the request, requesting permission, or by retrying the request. 4xx series errors other than 429 generally indicate an error with the request.

5xx

A 5xx series error indicates an issue with Lambda, or an issue with the function's configuration or resources. 5xx series errors can indicate a temporary condition that can be resolved without any action by the user. These issues can't be addressed by the invoking client or service, but a Lambda function's owner may be able to fix the issue.

For a complete list of invocation errors, see InvokeFunction errors.

Using the Lambda console

You can invoke your function on the Lambda console by configuring a test event and viewing the output. The output is captured in the function's execution logs and, when active tracing is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the Functions page of the Lambda console.
2. Choose the function to test, and choose Test.
3. Under Test event, select New event.
4. Select a Template.
5. For Name, enter a name for the test. In the text entry box, enter the JSON test event.
6. Choose Save changes.
7. Choose Test.

The Lambda console invokes your function synchronously and displays the result. To see the response, logs, and other information, expand the Details section.
Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- AWS Command Line Interface (AWS CLI) version 2
- AWS CLI – Quick configuration with `aws configure`

When you invoke a Lambda function in the AWS CLI, the AWS CLI splits the response into two documents. The AWS CLI response is displayed in your command prompt. If an error has occurred, the response contains a `FunctionError` field. The invocation response or error returned by the function is written to an output file. For example, `output.json` or `output.txt`.

The following `invoke` command example demonstrates how to invoke a function and write the invocation response to an `output.txt` file.

```
aws lambda invoke   
   --function-name my-function   
   --cli-binary-format raw-in-base64-out  
   --payload '{"key1": "value1", "key2": "value2", "key3": "value3"}' output.txt
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the AWS Command Line Interface User Guide for Version 2.

You should see the AWS CLI response in your command prompt:

```
{
   "StatusCode": 200,
   "FunctionError": "Unhandled",
   "ExecutedVersion": "$LATEST"
}
```

You should see the function invocation response in the `output.txt` file. In the same command prompt, you can also view the output in your command prompt using:

```
cat output.txt
```
You should see the invocation response in your command prompt.

```json
{"errorMessage": "'action'", "errorType": "KeyError", "stackTrace": [" File "/var/task/lambda_function.py", line 36, in lambda_handler
    result = ACTIONS[event['action']](event['number'])"]
}
```

Lambda also records up to 256 KB of the error object in the function’s logs. For more information, see [AWS Lambda function logging in Python](https://docs.aws.amazon.com/lambda/latest/dg/python-logging.html).

## Error handling in other AWS services

When another AWS service invokes your function, the service chooses the invocation type and retry behavior. AWS services can invoke your function on a schedule, in response to a lifecycle event on a resource, or to serve a request from a user. Some services invoke functions asynchronously and let Lambda handle errors, while others retry or pass errors back to the user.

For example, API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502 error code. To customize the error response, you must catch errors in your code and format a response in the required format.

We recommend using AWS X-Ray to determine the source of an error and its cause. X-Ray allows you to find out which component encountered an error, and see details about the errors. The following example shows a function error that resulted in a 502 response from API Gateway.

![Error handling in other AWS services](image)

For more information, see [Instrumenting Python code in AWS Lambda](https://docs.aws.amazon.com/lambda/latest/dg/instrument-python-code.html).

## Error examples

The following section shows common errors you may receive when creating, updating, or invoking your function using the Python [Lambda runtimes](https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html).
Example Runtime exception – ImportError

```
{
  "errorMessage": "Unable to import module 'lambda_function': Cannot import name
'_imaging' from 'PIL' (/var/task/PIL/__init__.py)",
  "errorType": "Runtime.ImportModuleError"
}
```

This error is a result of using the AWS Command Line Interface (AWS CLI) to upload a deployment package that contains a C or C++ library. For example, the Pillow (PIL), numpy, or pandas library.

We recommend using the AWS SAM CLI `sam build` command with the `--use-container` option to create your deployment package. Using the AWS SAM CLI with this option creates a Docker container with a Lambda-like environment that is compatible with Lambda.

Example JSON serialization error – Runtime.MarshalError

```
{
  "errorMessage": "Unable to marshal response: Object of type AttributeError is not
JSON serializable",
  "errorType": "Runtime.MarshalError"
}
```

This error can be the result of the base64-encoding mechanism you are using in your function code. For example:

```python
import base64
encrypted_data = base64.b64encode(payload_enc).decode("utf-8")
```

This error can also be the result of not specifying your `.zip` file as a binary file when you created or updated your function. We recommend using the `fileb://` command option to upload your deployment package (.zip file).

```
aws lambda create-function --function-name my-function --zip-file fileb://my-deployment-package.zip --handler lambda_function.lambda_handler --runtime python3.8 --role arn:aws:iam::your-account-id:role/lambda-ex
```
Sample applications

The GitHub repository for this guide includes sample applications that demonstrate the use of the errors. Each sample application includes scripts for easy deployment and cleanup, an AWS Serverless Application Model (AWS SAM) template, and supporting resources.

Sample Lambda applications in Python

- blank-python – A Python function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.

What's next?

- Learn how to show logging events for your Lambda function on the the section called “Logging” page.
Instrumenting Python code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of three SDK libraries:

- **AWS Distro for OpenTelemetry (ADOT)** – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- **AWS X-Ray SDK for Python** – An SDK for generating and sending trace data to X-Ray.
- **Powertools for AWS Lambda (Python)** – A developer toolkit to implement Serverless best practices and increase developer velocity.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

⚠️ Important

The X-Ray and Powertools for AWS Lambda SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see [Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs](#).

Sections

- [Using Powertools for AWS Lambda (Python) and AWS SAM for tracing](#)
- [Using Powertools for AWS Lambda (Python) and the AWS CDK for tracing](#)
- [Using ADOT to instrument your Python functions](#)
- [Using the X-Ray SDK to instrument your Python functions](#)
- [Activating tracing with the Lambda console](#)
- [Activating tracing with the Lambda API](#)
- [Activating tracing with AWS CloudFormation](#)
Using Powertools for AWS Lambda (Python) and AWS SAM for tracing

Follow the steps below to download, build, and deploy a sample Hello World Python application with integrated Powertools for AWS Lambda (Python) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Python 3.9
- AWS CLI version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World Python template.

   ```
   sam init --app-template hello-world-powertools-python --name sam-app --package-type Zip --runtime python3.9 --no-tracing
   ```

2. Build the app.

   ```
   cd sam-app && sam build
   ```

3. Deploy the app.

   ```
   sam deploy --guided
   ```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.
Note

For **HelloWorldFunction** may not have authorization defined, Is this okay?, make sure to enter y.

5. Get the URL of the deployed application:

```bash
aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```bash
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```json
{"message":"hello world"}
```

7. To get the traces for the function, run **sam traces**.

```bash
sam traces
```

The trace output looks like this:

```
New XRay Service Graph
 Start time: 2023-02-03 14:59:50+00:00
 End time: 2023-02-03 14:59:50+00:00
 Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [1]
 Summary_statistics:
 - total requests: 1
 - ok count(2XX): 1
 - error count(4XX): 0
 - fault count(5XX): 0
 - total response time: 0.924
 Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: []
 Summary_statistics:
 - total requests: 1
 - ok count(2XX): 1
```

Using Powertools for AWS Lambda (Python) and AWS SAM for tracing
- error count(4XX): 0
- fault count(5XX): 0
- total response time: 0.016

Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYt0c9j - Edges: [0]

Summary_statistics:
- total requests: 0
- ok count(2XX): 0
- error count(4XX): 0
- fault count(5XX): 0
- total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
- 0.924s - sam-app-HelloWorldFunction-YBg8yfYt0c9j [HTTP: 200]
- 0.016s - sam-app-HelloWorldFunction-YBg8yfYt0c9j
  - 0.739s - Initialization
  - 0.016s - Invocation
    - 0.013s - ## lambda_handler
    - 0.000s - ## app.hello
    - 0.000s - Overhead

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

⚠️ Note

You cannot configure the X-Ray sampling rate for your functions.

Using Powertools for AWS Lambda (Python) and the AWS CDK for tracing

Follow the steps below to download, build, and deploy a sample Hello World Python application with integrated Powertools for AWS Lambda (Python) modules using the AWS CDK. This
application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

**Prerequisites**

To complete the steps in this section, you must have the following:

- Python 3.9
- AWS CLI version 2
- AWS CDK version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

**Deploy a sample AWS CDK application**

1. Create a project directory for your new application.

   ```bash
   mkdir hello-world
   cd hello-world
   ```

2. Initialize the app.

   ```bash
   cdk init app --language python
   ```

3. Install the Python dependencies.

   ```bash
   pip install -r requirements.txt
   ```

4. Create a directory **lambda_function** under the root folder.

   ```bash
   mkdir lambda_function
   cd lambda_function
   ```

5. Create a file **app.py** and add the following code to the file. This is the code for the Lambda function.

   ```python
   from aws_lambda_powertools.event_handler import APIGatewayRestResolver
   ```
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count, value=1)

    # structured log
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/logger/
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.powertools.aws.dev/lambda-python/latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
# ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)

6. Open the **hello_world** directory. You should see a file called **hello_world_stack.py**.

    cd ..
    cd hello_world

7. Open **hello_world_stack.py** and add the following code to the file. This contains the Lambda Constructor, which creates the Lambda function, configures environment variables for
Powertools and sets log retention to one week, and the [ApiGatewayv1 Constructor](https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-load-balancer-concepts.html), which creates the REST API.

```python
from aws_cdk import (  
    Stack,  
    aws_apigateway as apigwv1,  
    aws_lambda as lambda_,  
    CfnOutput,  
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,  
            id="lambda-powertools",  
            # At the moment we wrote this example, the aws_lambda_python_alpha CDK
            # constructor is in Alpha, so we use layer to make the example simpler
            # See https://docs.aws.amazon.com/cdk/api/v2/python/
            # Check all Powertools layers versions here: https://
            # docs.powertools.aws.dev/lambda-python/latest/#lambda-layer
            layer_version_arn=f"arn:aws:lambda:
              {self.region}:017000801446:layer:AWSLambdaPowertoolsPythonV2:21"
        )

        function = lambda_.Function(self,
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_9,
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
            memory_size=128,
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
            }
        )
```

Using Powertools for AWS Lambda (Python) and the AWS CDK for tracing
8. Deploy your application.

cd ..
cdk deploy

9. Get the URL of the deployed application:

```bash
aws cloudformation describe-stacks --stack-name HelloWorldStack --query 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. Invoke the API endpoint:

```bash
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```json
{"message":"hello world"}
```

11. To get the traces for the function, run `sam traces`.

```bash
sam traces
```

The traces output looks like this:

```
New XRay Service Graph
Start time: 2023-02-03 14:59:50+00:00
End time: 2023-02-03 14:59:50+00:00
Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYt0c9j - Edges: [1]
```
Summary_statistics:
- total requests: 1
- ok count(2XX): 1
- error count(4XX): 0
- fault count(5XX): 0
- total response time: 0.924
Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYt0c9j
- Edges: []

Summary_statistics:
- total requests: 1
- ok count(2XX): 1
- error count(4XX): 0
- fault count(5XX): 0
- total response time: 0.016
Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYt0c9j - Edges: [0]

Summary_statistics:
- total requests: 0
- ok count(2XX): 0
- error count(4XX): 0
- fault count(5XX): 0
- total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
- 0.924s - sam-app-HelloWorldFunction-YBg8yfYt0c9j [HTTP: 200]
- 0.016s - sam-app-HelloWorldFunction-YBg8yfYt0c9j
  - 0.739s - Initialization
  - 0.016s - Invocation
    - 0.013s - ## lambda_handler
      - 0.000s - ## app.hello
    - 0.000s - Overhead

12. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

cdk destroy

Using ADOT to instrument your Python functions

ADOT provides fully managed Lambda layers that package everything you need to collect telemetry data using the OTel SDK. By consuming this layer, you can instrument your Lambda
functions without having to modify any function code. You can also configure your layer to do custom initialization of OTel. For more information, see Custom configuration for the ADOT Collector on Lambda in the ADOT documentation.

For Python runtimes, you can add the AWS managed Lambda layer for ADOT Python to automatically instrument your functions. This layer works for both arm64 and x86_64 architectures. For detailed instructions on how to add this layer, see AWS Distro for OpenTelemetry Lambda Support for Python in the ADOT documentation.

Using the X-Ray SDK to instrument your Python functions

To record details about calls that your Lambda function makes to other resources in your application, you can also use the AWS X-Ray SDK for Python. To get the SDK, add the aws-xray-sdk package to your application's dependencies.

Example requirements.txt

```bash
jsonpickle==1.3
aws-xray-sdk==2.4.3
```

In your function code, you can instrument AWS SDK clients by patching the boto3 library with the aws_xray_sdk.core module.

Example function – Tracing an AWS SDK client

```python
import boto3
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

logger = logging.getLogger()
logger.setLevel(logging.INFO)
patch_all()

client = boto3.client('lambda')
client.get_account_settings()

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES
' +jsonpickle.encode(dict(**os.environ)))
...
After you add the correct dependencies and make the necessary code changes, activate tracing in your function's configuration via the Lambda console or the API.

Activating tracing with the Lambda console

To toggle active tracing on your Lambda function with the console, follow these steps:

**To turn on active tracing**

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring and operations tools**.
4. Choose **Edit**.
5. Under **X-Ray**, toggle on **Active tracing**.
6. Choose **Save**.

Activating tracing with the Lambda API

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- [UpdateFunctionConfiguration](#)
- [GetFunctionConfiguration](#)
- [CreateFunction](#)

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.
Activating tracing with AWS CloudFormation

To activate tracing on an AWS::Lambda::Function resource in an AWS CloudFormation template, use the TracingConfig property.

Example **function-inline.yml** – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
...
```

For an AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function resource, use the Tracing property.

Example **template.yml** – Tracing configuration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
...
```

Interpreting an X-Ray trace

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's execution role. Otherwise, add the AWSXRayDaemonWriteAccess policy to the execution role.

After you've configured active tracing, you can observe specific requests through your application. The X-Ray service graph shows information about your application and all its components. The following example from the error processor sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.
X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

Note
You cannot configure the X-Ray sampling rate for your functions.

When using active tracing, Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes for the primary function from the error processor sample application.

The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace...
with these two segments. Both are named **my-function**, but one has an origin of `AWS::Lambda` and the other has origin `AWS::Lambda::Function`.

This example expands the function segment to show its three subsegments:

- **Initialization** – Represents time spent loading your function and running initialization code. This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see the [AWS X-Ray SDK for Python](https://docs.aws.amazon.com/xray/latest/devguide/x-ray-sdk-python.html) in the AWS X-Ray Developer Guide.

### Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see [AWS X-Ray pricing](https://aws.amazon.com/xray/pricing/).

### Storing runtime dependencies in a layer (X-Ray SDK)

If you use the X-Ray SDK to instrument AWS SDK clients your function code, your deployment package can become quite large. To avoid uploading runtime dependencies every time you update your function code, package the X-Ray SDK in a [Lambda layer](https://docs.aws.amazon.com/lambda/latest/dg/configuration-lambda-layer.html).
The following example shows an AWS::Serverless::LayerVersion resource that stores the AWS X-Ray SDK for Python.

Example [template.yml](#) – Dependencies layer

_resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
        ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-python-lib
      Description: Dependencies for the blank-python sample app.
      ContentUri: package/.
      CompatibleRuntimes:
        - python3.8

With this configuration, you update the library layer only if you change your runtime dependencies. Since the function deployment package contains only your code, this can help reduce upload times.

Creating a layer for dependencies requires build changes to generate the layer archive prior to deployment. For a working example, see the [blank-python](#) sample application.
Building Lambda functions with Ruby

You can run Ruby code in AWS Lambda. Lambda provides runtimes for Ruby that run your code to process events. Your code runs in an environment that includes the AWS SDK for Ruby, with credentials from an AWS Identity and Access Management (IAM) role that you manage.

Lambda supports the following Ruby runtimes.

Ruby

<table>
<thead>
<tr>
<th>Name</th>
<th>Identifier</th>
<th>SDK</th>
<th>Operating system</th>
<th>Deprecation date</th>
<th>Block function create</th>
<th>Block function update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ruby 3.2</td>
<td>ruby3.2</td>
<td>3.1.0</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To create a Ruby function

1. Open the Lambda console.
2. Choose Create function.
3. Configure the following settings:
   - Function name: Enter a name for the function.
   - Runtime: Choose Ruby 3.2.
4. Choose Create function.
5. To configure a test event, choose Test.
6. For Event name, enter test.
7. Choose Save changes.
8. To invoke the function, choose Test.

The console creates a Lambda function with a single source file named lambda_function.rb. You can edit this file and add more files in the built-in code editor. To save your changes, choose Save. Then, to run your code, choose Test.
Note

The Lambda console uses AWS Cloud9 to provide an integrated development environment in the browser. You can also use AWS Cloud9 to develop Lambda functions in your own environment. For more information, see Working with AWS Lambda functions using the AWS Toolkit in the AWS Cloud9 user guide.

The lambda_function.rb file exports a function named lambda_handler that takes an event object and a context object. This is the handler function that Lambda calls when the function is invoked. The Ruby function runtime gets invocation events from Lambda and passes them to the handler. In the function configuration, the handler value is lambda_function.lambda_handler.

When you save your function code, the Lambda console creates a .zip file archive deployment package. When you develop your function code outside of the console (using an IDE) you need to create a deployment package to upload your code to the Lambda function.

Note

To get started with application development in your local environment, deploy one of the sample applications available in this guide's GitHub repository.

Sample Lambda applications in Ruby

- blank-ruby – A Ruby function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.
- Ruby Code Samples for AWS Lambda – Code samples written in Ruby that demonstrate how to interact with AWS Lambda.

The function runtime passes a context object to the handler, in addition to the invocation event. The context object contains additional information about the invocation, the function, and the execution environment. More information is available from environment variables.

Your Lambda function comes with a CloudWatch Logs log group. The function runtime sends details about each invocation to CloudWatch Logs. It relays any logs that your function outputs during invocation. If your function returns an error, Lambda formats the error and returns it to the invoker.
Enabling Yet Another Ruby JIT (YJIT)

The Ruby 3.2 runtime supports YJIT, a lightweight, minimalistic Ruby JIT compiler. YJIT provides significantly higher performance, but also uses more memory than the Ruby interpreter. YJIT is recommended for Ruby on Rails workloads.

YJIT is not enabled by default. To enable YJIT for a Ruby 3.2 function, set the RUBY_YJIT_ENABLE environment variable to 1. To confirm that YJIT is enabled, print the result of the RubyVM::YJIT.enabled? method.

Example — Confirm that YJIT is enabled

```ruby
puts(RubyVM::YJIT.enabled?)
# => true
```

Topics

- AWS Lambda function handler in Ruby
- Working with .zip file archives for Ruby Lambda functions
- Deploy Ruby Lambda functions with container images
- AWS Lambda context object in Ruby
- AWS Lambda function logging in Ruby
- AWS Lambda function errors in Ruby
- Instrumenting Ruby code in AWS Lambda
AWS Lambda function handler in Ruby

The Lambda function handler is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

In the following example, the file function.rb defines a handler method named handler. The handler function takes two objects as input and returns a JSON document.

Example function.rb

```ruby
require 'json'

def handler(event:, context:)
  { event: JSON.generate(event), context: JSON.generate(context.inspect) }
end
```

In your function configuration, the handler setting tells Lambda where to find the handler. For the preceding example, the correct value for this setting is function.handler. It includes two names separated by a dot: the name of the file and the name of the handler method.

You can also define your handler method in a class. The following example defines a handler method named process on a class named Handler in a module named LambdaFunctions.

Example source.rb

```ruby
module LambdaFunctions
  class Handler
    def self.process(event:,context:)
      "Hello!"
    end
  end
end
```

In this case, the handler setting is source.LambdaFunctions::Handler::process.

The two objects that the handler accepts are the invocation event and context. The event is a Ruby object that contains the payload that's provided by the invoker. If the payload is a JSON document, the event object is a Ruby hash. Otherwise, it's a string. The context object has methods and properties that provide information about the invocation, the function, and the execution environment.
The function handler is executed every time your Lambda function is invoked. Static code outside of the handler is executed once per instance of the function. If your handler uses resources like SDK clients and database connections, you can create them outside of the handler method to reuse them for multiple invocations.

Each instance of your function can process multiple invocation events, but it only processes one event at a time. The number of instances processing an event at any given time is your function's concurrency. For more information about the Lambda execution environment, see Lambda execution environment.
Working with .zip file archives for Ruby Lambda functions

Your AWS Lambda function's code comprises a .rb file containing your function's handler code, together with any additional dependencies (gems) your code depends on. To deploy this function code to Lambda, you use a *deployment package*. This package may either be a .zip file archive or a container image. For more information about using container images with Ruby, see Deploy Ruby Lambda functions with container images.

To create your deployment package as .zip file archive, you can use your command-line tool’s built-in .zip file archive utility, or any other .zip file utility such as 7zip. The examples shown in the following sections assume you're using a command-line `zip` tool in a Linux or MacOS environment. To use the same commands in Windows, you can install the Windows Subsystem for Linux to get a Windows-integrated version of Ubuntu and Bash.

Note that Lambda uses POSIX file permissions, so you may need to set permissions for the deployment package folder before you create the .zip file archive.

The example commands in the following sections use the Bundler utility to add dependencies to your deployment package. To install bundler, run the following command.

```
gem install bundler
```

### Sections

- Dependencies in Ruby
- Creating a .zip deployment package with no dependencies
- Creating a .zip deployment packaged with dependencies
- Creating a Ruby layer for your dependencies
- Creating .zip deployment packages with native libraries
- Creating and updating Ruby Lambda functions using .zip files

### Dependencies in Ruby

For Lambda functions that use the Ruby runtime, a dependency can be any Ruby gem. When you deploy your function using a .zip archive, you can either add these dependencies to your .zip file with your function code or use a Lambda layer. A layer is a separate .zip file that can contain additional code and other content. To learn more about using Lambda layers, see Lambda layers.
Versions 2.7 and 3.2 of the Ruby runtime includes the AWS SDK for Ruby. If your function uses the SDK, you don't need to bundle it with your code. However, to maintain full control of your dependencies, or to use a specific version of the SDK, you can add it to your function's deployment package. You can either include the SDK in your .zip file, or add it using a Lambda layer. Dependencies in your .zip file or in Lambda layers take precedence over versions included in the runtime. To see which version of the SDK for Ruby is included in your runtime version, see [Lambda runtimes](#).

Under the [AWS shared responsibility model](#), you are responsible for the management of any dependencies in your functions' deployment packages. This includes applying updates and security patches. To update dependencies in your function's deployment package, first create a new .zip file and then upload it to Lambda. See [Creating a .zip deployment packaged with dependencies](#) and [Creating and updating Ruby Lambda functions using .zip files](#) for more information.

### Creating a .zip deployment package with no dependencies

If your function code has no dependencies, your .zip file contains only the .rb file with your function's handler code. Use your preferred zip utility to create a .zip file with your .rb file at the root. If the .rb file is not at the root of your .zip file, Lambda won’t be able to run your code.

To learn how to deploy your .zip file to create a new Lambda function or update an existing one, see [Creating and updating Ruby Lambda functions using .zip files](#).

### Creating a .zip deployment packaged with dependencies

If your function code depends on additional Ruby gems, you can either add these dependencies to your .zip file with your function code or use a [Lambda layer](#). The instructions in this section show you how to include dependencies in your .zip deployment package. For instructions on how to include your dependencies in a layer, see the section called “Creating a Ruby layer for your dependencies”.

Suppose your function code is saved in a file named `lambda_function.rb` in your project directory. The following example CLI commands create a .zip file named `my_deployment_package.zip` containing your function code and its dependencies.

**To create the deployment package**

1. In your project directory, create a Gemfile to specify your dependencies in.
2. Using your preferred text editor, edit the Gemfile to specify your function's dependencies. For example, to use the TZInfo gem, edit your Gemfile to look like the following.

   ```ruby
   source "https://rubygems.org"
   gem "tzinfo"
   ```

3. Run the following command to install the gems specified in your Gemfile in your project directory. This command sets `vendor/bundle` as the default path for gem installations.

   ```bash
   bundle config set --local path 'vendor/bundle' && bundle install
   ```

   You should see output similar to the following.

   ```
   Fetching gem metadata from https://rubygems.org/............
   Resolving dependencies...
   Using bundler 2.4.13
   Fetching tzinfo 2.0.6
   Installing tzinfo 2.0.6
   ...
   ```

   **Note**

   To install gems globally again later, run the following command.

   ```bash
   bundle config set --local system 'true'
   ```

4. Create a .zip file archive containing the `lambda_function.rb` file with your function's handler code and the dependencies you installed in the previous step.

   ```bash
   zip -r my_deployment_package.zip lambda_function.rb vendor
   ```

   You should see output similar to the following.

   ```
   adding: lambda_function.rb (deflated 37%)
   adding: vendor/ (stored 0%)
   adding: vendor/bundle/ (stored 0%)
   ```
Creating a Ruby layer for your dependencies

The instructions in this section show you how to include your dependencies in a layer. For instructions on how to include your dependencies in your deployment package, see the section called "Creating a .zip deployment packaged with dependencies".

When you add a layer to a function, Lambda loads the layer content into the /opt directory of that execution environment. For each Lambda runtime, the PATH variable already includes specific folder paths within the /opt directory. To ensure that your layer content gets picked up by the PATH variable, include the content in the following folder paths:

- ruby/gems/2.7.0 (GEM_PATH)
- ruby/lib (RUBYLIB)

For example, your layer .zip file structure might look like the following:

```text
json.zip
  # ruby/gems/2.7.0/
  | build_info
  | cache
  | doc
  | extensions
  | gems
  | # json-2.1.0
  # specifications
  # json-2.1.0.gemspec
```

In addition, Lambda automatically detects any libraries in the /opt/lib directory, and any binaries in the /opt/bin directory. To ensure that Lambda properly finds your layer content, you can also create a layer with the following structure:

```text
custom-layer.zip
```
After you package your layer, see the section called “Creating and deleting layers” and the section called “Adding layers” to complete your layer setup.

**Creating .zip deployment packages with native libraries**

Many common Ruby gems such as nokogiri,.nio4r, and mysql contain native extensions written in C. When you add libraries containing C code to your deployment package, you must build your package correctly to ensure that it’s compatible with the Lambda execution environment.

For production applications, we recommend building and deploying your code using the AWS Serverless Application Model (AWS SAM). In AWS SAM use the `sam build --use-container` option to build your function inside a Lambda-like Docker container. To learn more about using AWS SAM to deploy your function code, see Building applications in the AWS SAM Developer Guide.

To create a .zip deployment package containing gems with native extensions without using AWS SAM, you can alternatively use a container to bundle your dependencies in an environment that is the same as the Lambda Ruby runtime environment. To complete these steps, you must have Docker installed on your build machine. To learn more about installing Docker, see Install Docker Engine.

**To create a .zip deployment package in a Docker container**

1. Create a folder on your local build machine to save your container in. Inside that folder, create a file named `dockerfile` and paste the following code into it.

   ```
   FROM public.ecr.aws/sam/build-ruby3.2:latest-x86_64
   RUN gem update bundler
   CMD "/bin/bash"
   ```

2. Inside the folder you created your `dockerfile` in, run the following command to create the Docker container.

   ```
   docker build -t awsruby32 .
   ```
3. Navigate to the project directory containing the `.rb` file with your function's handler code and the `Gemfile` specifying your function's dependencies. From inside that directory, run the following command to start the Lambda Ruby container.

**Linux/MacOS**

```bash
docker run --rm -it -v $PWD:/var/task -w /var/task awsruby32
```

**Note**

In MacOS, you might see a warning informing you that the requested image's platform does not match the detected host platform. Ignore this warning.

**Windows PowerShell**

```bash
docker run --rm -it -v ${pwd}:var/task -w /var/task awsruby32
```

When your container starts, you should see a bash prompt.

**bash-4.2#**

4. Configure the bundle utility to install the gems specified in your `Gemfile` in a local `vendor/bundle` directory and install your dependencies.

**bash-4.2#**

```bash
bundle config set --local path 'vendor/bundle' && bundle install
```

5. Create the `.zip` deployment package with your function code and its dependencies. In this example, the file containing your function's handler code is named `lambda_function.rb`.

**bash-4.2#**

```bash
zip -r my_deployment_package.zip lambda_function.rb vendor
```

6. Exit the container and return to your local project directory.

**bash-4.2#**

```bash
exit
```

You can now use the `.zip` file deployment package to create or update your Lambda function. See [Creating and updating Ruby Lambda functions using .zip files](#).
Creating and updating Ruby Lambda functions using .zip files

Once you have created your .zip deployment package, you can use it to create a new Lambda function or update an existing one. You can deploy your .zip package using the Lambda console, the AWS Command Line Interface, and the Lambda API. You can also create and update Lambda functions using AWS Serverless Application Model (AWS SAM) and AWS CloudFormation.

The maximum size for a .zip deployment package for Lambda is 250 MB (unzipped). Note that this limit applies to the combined size of all the files you upload, including any Lambda layers.

Creating and updating functions with .zip files using the console

To create a new function, you must first create the function in the console, then upload your .zip archive. To update an existing function, open the page for your function, then follow the same procedure to add your updated .zip file.

If your .zip file is less than 50MB, you can create or update a function by uploading the file directly from your local machine. For .zip files greater than 50MB, you must upload your package to an Amazon S3 bucket first. For instructions on how to upload a file to an Amazon S3 bucket using the AWS Management Console, see Getting started with Amazon S3. To upload files using the AWS CLI, see Move objects in the AWS CLI User Guide.

⚠️ Note

You cannot convert an existing container image function to use a .zip archive. You must create a new function.

To create a new function (console)

1. Open the Functions page of the Lambda console and choose Create Function.
2. Choose Author from scratch.
3. Under Basic information, do the following:
   a. For Function name, enter the name for your function.
   b. For Runtime, select the runtime you want to use.
   c. (Optional) For Architecture, choose the instruction set architecture for your function. The default architecture is x86_64. Ensure that the .zip deployment package for your function is compatible with the instruction set architecture you select.
4. (Optional) Under Permissions, expand Change default execution role. You can create a new Execution role or use an existing one.

5. Choose Create function. Lambda creates a basic 'Hello world' function using your chosen runtime.

To upload a .zip archive from your local machine (console)

1. In the Functions page of the Lambda console, choose the function you want to upload the .zip file for.
2. Select the Code tab.
3. In the Code source pane, choose Upload from.
5. To upload the .zip file, do the following:
   a. Select Upload, then select your .zip file in the file chooser.
   b. Choose Open.
   c. Choose Save.

To upload a .zip archive from an Amazon S3 bucket (console)

1. In the Functions page of the Lambda console, choose the function you want to upload a new .zip file for.
2. Select the Code tab.
3. In the Code source pane, choose Upload from.
4. Choose Amazon S3 location.
5. Paste the Amazon S3 link URL of your .zip file and choose Save.

Updating .zip file functions using the console code editor

For some functions with .zip deployment packages, you can use the Lambda console’s built-in code editor to update your function code directly. To use this feature, your function must meet the following criteria:

- Your function must use one of the interpreted language runtimes (Python, Node.js, or Ruby)
- Your function’s deployment package must be smaller than 3MB.
Function code for functions with container image deployment packages cannot be edited directly in the console.

**To update function code using the console code editor**

1. Open the [Functions page](#) of the Lambda console and select your function.
2. Select the **Code** tab.
3. In the **Code source** pane, select your source code file and edit it in the integrated code editor.
4. When you have finished editing your code, choose **Deploy** to save your changes and update your function.

**Creating and updating functions with .zip files using the AWS CLI**

You can use the [AWS CLI](#) to create a new function or to update an existing one using a .zip file. Use the `create-function` and `update-function-code` commands to deploy your .zip package. If your .zip file is smaller than 50MB, you can upload the .zip package from a file location on your local build machine. For larger files, you must upload your .zip package from an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see [Move objects](#) in the [AWS CLI User Guide](#).

⚠️ **Note**

If you upload your .zip file from an Amazon S3 bucket using the AWS CLI, the bucket must be located in the same AWS Region as your function.

To create a new function using a .zip file with the AWS CLI, you must specify the following:

- The name of your function (`--function-name`)
- Your function’s runtime (`--runtime`)
- The Amazon Resource Name (ARN) of your function’s execution role (`--role`)
- The name of the handler method in your function code (`--handler`)

You must also specify the location of your .zip file. If your .zip file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.
aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip

To specify the location of .zip file in an Amazon S3 bucket, use the --code option as shown in the following example command. You only need to use the S3ObjectVersion parameter for versioned objects.

aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=myBucketName,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion

To update an existing function using the CLI, you specify the the name of your function using the --function-name parameter. You must also specify the location of the .zip file you want to use to update your function code. If your .zip file is located in a folder on your local build machine, use the --zip-file option to specify the file path, as shown in the following example command.

aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip

To specify the location of .zip file in an Amazon S3 bucket, use the --s3-bucket and --s3-key options as shown in the following example command. You only need to use the --s3-object-version parameter for versioned objects.

aws lambda update-function-code --function-name myFunction \
--s3-bucket myBucketName --s3-key myFileName.zip --s3-object-version myObjectVersion

Creating and updating functions with .zip files using the Lambda API

To create and update functions using a .zip file archive, use the following API operations:

- `CreateFunction`
- `UpdateFunctionCode`
Creating and updating functions with .zip files using AWS SAM

The AWS Serverless Application Model (AWS SAM) is a toolkit that helps streamline the process of building and running serverless applications on AWS. You define the resources for your application in a YAML or JSON template and use the AWS SAM command line interface (AWS SAM CLI) to build, package, and deploy your applications. When you build a Lambda function from an AWS SAM template, AWS SAM automatically creates a .zip deployment package or container image with your function code and any dependencies you specify. To learn more about using AWS SAM to build and deploy Lambda functions, see Getting started with AWS SAM in the AWS Serverless Application Model Developer Guide.

You can also use AWS SAM to create a Lambda function using an existing .zip file archive. To create a Lambda function using AWS SAM, you can save your .zip file in an Amazon S3 bucket or in a local folder on your build machine. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the AWS CLI User Guide.

In your AWS SAM template, the AWS::Serverless::Function resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

- PackageType - set to Zip
- CodeUri - set to the function code's Amazon S3 URI, path to local folder, or FunctionCode object
- Runtime - Set to your chosen runtime

With AWS SAM, if your .zip file is larger than 50MB, you don’t need to upload it to an Amazon S3 bucket first. AWS SAM can upload .zip packages up to the maximum allowed size of 250MB (unzipped) from a location on your local build machine.

To learn more about deploying functions using .zip file in AWS SAM, see AWS::Serverless::Function in the AWS SAM Developer Guide.

Creating and updating functions with .zip files using AWS CloudFormation

You can use AWS CloudFormation to create a Lambda function using a .zip file archive. To create a Lambda function from a .zip file, you must first upload your file to an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the AWS CLI User Guide.
In your AWS CloudFormation template, the AWS::Lambda::Function resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

- **PackageType** - Set to `Zip`
- **Code** - Enter the Amazon S3 bucket name and the .zip file name in the `S3Bucket` and `S3Key` fields
- **Runtime** - Set to your chosen runtime

Deploy Ruby Lambda functions with container images

There are three ways to build a container image for a Ruby Lambda function:

- **Using an AWS base image for Ruby**

  The AWS base images are preloaded with a language runtime, a runtime interface client to manage the interaction between Lambda and your function code, and a runtime interface emulator for local testing.

- **Using an AWS OS-only base image**

  AWS OS-only base images contain an Amazon Linux distribution and the runtime interface emulator. These images are commonly used to create container images for compiled languages, such as Go and Rust, and for a language or language version that Lambda doesn't provide a base image for, such as Node.js 19. You can also use OS-only base images to implement a custom runtime. To make the image compatible with Lambda, you must include the runtime interface client for Ruby in the image.

- **Using a non-AWS base image**

  You can use an alternative base image from another container registry, such as Alpine Linux or Debian. You can also use a custom image created by your organization. To make the image compatible with Lambda, you must include the runtime interface client for Ruby in the image.

**Tip**

To reduce the time it takes for Lambda container functions to become active, see Use multi-stage builds in the Docker documentation. To build efficient container images, follow the Best practices for writing Dockerfiles.

This page explains how to build, test, and deploy container images for Lambda.

**Topics**

- AWS base images for Ruby
- Using an AWS base image for Ruby
- Using an alternative base image with the runtime interface client
AWS provides the following base images for Ruby:

<table>
<thead>
<tr>
<th>Tags</th>
<th>Runtime</th>
<th>Operating system</th>
<th>Dockerfile</th>
<th>Deprecation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Ruby 3.2</td>
<td>Amazon Linux 2</td>
<td>Dockerfile for Ruby 3.2 on GitHub</td>
<td></td>
</tr>
</tbody>
</table>

Amazon ECR repository: [gallery.ecr.aws/lambda/ruby](https://gallery.ecr.aws/lambda/ruby)

### Using an AWS base image for Ruby

#### Prerequisites

To complete the steps in this section, you must have the following:

- AWS Command Line Interface (AWS CLI) version 2
- Docker
- Ruby

#### Creating an image from a base image

**To create a container image for Ruby**

1. Create a directory for the project, and then switch to that directory.
   
   ```bash
   mkdir example
cd example
   ```

2. Create a new file called `Gemfile`. This is where you list your application's required RubyGems packages. The AWS SDK for Ruby is available from RubyGems. You should choose specific AWS service gems to install. For example, to use the [Ruby gem for Lambda](https), your Gemfile should look like this:
   
   ```ruby
   source 'https://rubygems.org'
   ```
Alternatively, the `aws-sdk` gem contains every available AWS service gem. This gem is very large. We recommend that you use it only if you depend on many AWS services.

3. Install the dependencies specified in the Gemfile using `bundle install`.

4. Create a new file called `lambda_function.rb`. You can add the following sample function code to the file for testing, or use your own.

**Example Ruby function**

```ruby
module LambdaFunction
  class Handler
    def self.process(event:, context:)
      "Hello from Lambda!"
    end
  end
end
```

5. Create a new Dockerfile. The following is an example Dockerfile that uses an AWS base image. This Dockerfile uses the following configuration:

- Set the `FROM` property to the URI of the base image.
- Use the `COPY` command to copy the function code and runtime dependencies to `{LAMBDA_TASK_ROOT}`.
- Set the `CMD` argument to the Lambda function handler.

**Example Dockerfile**

```bash
FROM public.ecr.aws/lambda/ruby:3.2

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
    bundle config set --local path 'vendor/bundle' && \
    bundle install
```
# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set the CMD to your handler (could also be done as a parameter override outside of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]

6. Build the Docker image with the `docker build` command. The following example names the image `docker-image` and gives it the test tag.

```bash
docker build --platform linux/amd64 -t docker-image:test .
```

**Note**
The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

1. Start the Docker image with the `docker run` command. In this example, `docker-image` is the image name and `test` is the tag.

```bash
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

**Note**
If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

2. From a new terminal window, post an event to the local endpoint.
Linux/macOS

In Linux and macOS, run the following `curl` command:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{"payload":"hello world!"}"
```

PowerShell

In PowerShell, run the following `Invoke-WebRequest` command:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{}' -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{"payload":"hello world!"}'' -ContentType "application/json"
```

3. Get the container ID.

```
docker ps
```

4. Use the `docker kill` command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```
docker kill 3766c4ab331c
```
Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the `get-login-password` command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace `111122223333` with your AWS account ID.

   ```bash
   aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
   ```

2. Create a repository in Amazon ECR using the `create-repository` command.

   ```bash
   aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
   ```

   **Note**
   The Amazon ECR repository must be in the same AWS Region as the Lambda function.

   If successful, you see a response like this:

   ```json
   {
       "repository": {
           "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
           "registryId": "111122223333",
           "repositoryName": "hello-world",
           "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
           "createdAt": "2023-03-09T10:39:01+00:00",
           "imageTagMutability": "MUTABLE",
           "imageScanningConfiguration": {
               "scanOnPush": true
           },
           "encryptionConfiguration": {
           
```
3. Copy the repositoryUri from the output in the previous step.

4. Run the `docker tag` command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - Replace `docker-image:test` with the name and `tag` of your Docker image.
   - Replace `<ECRrepositoryUri>` with the repositoryUri that you copied. Make sure to include `:latest` at the end of the URI.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Example:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. Run the `docker push` command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

   ```
   aws lambda create-function
   --function-name hello-world
   --package-type Image
   --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   --role arn:aws:iam::111122223333:role/lambda-ex
   ```
8. Invoke the function.

aws lambda invoke --function-name hello-world response.json

You should see a response like this:

```json
{
   "ExecutedVersion": "$LATEST",
   "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the `update-function-code` command to deploy the image to the Lambda function.

### Using an alternative base image with the runtime interface client

If you use an OS-only base image or an alternative base image, you must include the runtime interface client in your image. The runtime interface client extends the Lambda runtime API, which manages the interaction between Lambda and your function code.

Install the [Lambda runtime interface client for Ruby](https://rubygems.org/) using the RubyGems.org package manager:

```bash
gem install aws_lambda_ric
```

You can also download the Ruby runtime interface client from GitHub. The runtime interface client supports Ruby versions 2.5.x to 2.7.x.

The following example demonstrates how to build a container image for Ruby using a non-AWS base image. The example Dockerfile uses an official Ruby base image. The Dockerfile includes the runtime interface client.
Prerequisites

To complete the steps in this section, you must have the following:

- AWS Command Line Interface (AWS CLI) version 2
- Docker
- Ruby

Creating an image from an alternative base image

To create a container image for Ruby using an alternative base image

1. Create a directory for the project, and then switch to that directory.

   ```bash
   mkdir example
cd example
   ```

2. Create a new file called `Gemfile`. This is where you list your application's required RubyGems packages. The AWS SDK for Ruby is available from RubyGems. You should choose specific AWS service gems to install. For example, to use the Ruby gem for Lambda, your Gemfile should look like this:

   ```ruby
   source 'https://rubygems.org'
gem 'aws-sdk-lambda'
   ```

   Alternatively, the `aws-sdk` gem contains every available AWS service gem. This gem is very large. We recommend that you use it only if you depend on many AWS services.

3. Install the dependencies specified in the Gemfile using `bundle install`.

   ```bash
   bundle install
   ```

4. Create a new file called `lambda_function.rb`. You can add the following sample function code to the file for testing, or use your own.

   **Example Ruby function**

   ```ruby
   module LambdaFunction
   class Handler
   ```
5. Create a new Dockerfile. The following Dockerfile uses a Ruby base image instead of an AWS base image. The Dockerfile includes the runtime interface client for Ruby, which makes the image compatible with Lambda. Alternatively, you can add the runtime interface client to your application's Gemfile.

- Set the FROM property to the Ruby base image.
- Use the COPY command to copy the function code and runtime dependencies to {LAMBDA_TASK_ROOT}.
- Set the ENTRYPOINT to the module that you want the Docker container to run when it starts. In this case, the module is the runtime interface client.
- Set the CMD argument to the Lambda function handler.

**Example Dockerfile**

```dolang
FROM ruby:2.7

# Install the runtime interface client for Ruby
RUN gem install aws_lambda_ric

# Add the runtime interface client to the PATH
ENV PATH="/usr/local/bundle/bin:${PATH}"

# Create a directory for the Lambda function
ENV LAMBDA_TASK_ROOT=/var/task
RUN mkdir -p ${LAMBDA_TASK_ROOT}
WORKDIR ${LAMBDA_TASK_ROOT}

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && 
    bundle config set --local path 'vendor/bundle' && 
    bundle install
```
# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "aws_lambda_ric" ]

# Set the CMD to your handler (could also be done as a parameter override outside
# of the Dockerfile)
CMD [ "lambda_function::LambdaFunction::Handler::process" ]

6. Build the Docker image with the `docker build` command. The following example names the
image `docker-image` and gives it the test `tag`.

docker build --platform linux/amd64 -t docker-image:test.

**Note**
The command specifies the `--platform linux/amd64` option to ensure that your
container is compatible with the Lambda execution environment regardless of the
architecture of your build machine. If you intend to create a Lambda function using
the ARM64 instruction set architecture, be sure to change the command to use the `--
platform linux/arm64` option instead.

(Optional) Test the image locally

Use the runtime interface emulator to locally test the image. You can build the emulator into your
image or install it on your local machine.

To install and run the runtime interface emulator on your local machine

1. From your project directory, run the following command to download the runtime interface
emulator (x86-64 architecture) from GitHub and install it on your local machine.

   Linux/macOS

   ```bash
   mkdir -p ~/.aws-lambda-rie && \
   ```
To install the arm64 emulator, replace the GitHub repository URL in the previous command with the following:

https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}
$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

To install the arm64 emulator, replace the $downloadLink with the following:

https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64

2. Start the Docker image with the **docker run** command. Note the following:

   • docker-image is the image name and test is the tag.
   • aws_lambda_ric lambda_function.LambdaFunction::Handler.process is the ENTRYPOINT followed by the CMD from your Dockerfile.

Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
    aws_lambda_ric lambda_function.LambdaFunction::Handler.process
```
PowerShell

```bash
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p 9000:8080 `--entrypoint /aws-lambda/aws-lambda-rie `docker-image:test `aws_lambda_ric lambda_function.LambdaFunction::Handler.process
```

This command runs the image as a container and creates a local endpoint at localhost:9000/2015-03-31/functions/function/invocations.

Note

If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

3. Post an event to the local endpoint.

Linux/macOS

In Linux and macOS, run the following curl command:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{"payload":"hello world!"}'
```

PowerShell

In PowerShell, run the following Invoke-WebRequest command:

```bash
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{}' -ContentType "application/json"
```
This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType "application/json"
```

4. Get the container ID.
```
docker ps
```

5. Use the `docker kill` command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.
```
docker kill 3766c4ab331c
```

## Deploying the image

### To upload the image to Amazon ECR and create the Lambda function

1. Run the `get-login-password` command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace `111122223333` with your AWS account ID.

   ```
   aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
   ```

2. Create a repository in Amazon ECR using the `create-repository` command.

   ```
   aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
   ```

Using a non-AWS base image
Note

The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```json
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copy the repositoryUri from the output in the previous step.

4. Run the `docker tag` command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - Replace `docker-image:test` with the name and tag of your Docker image.
   - Replace `<ECRrepositoryUri>` with the repositoryUri that you copied. Make sure to include :latest at the end of the URI.

   ```bash
docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Example:
5. Run the `docker push` command to deploy your local image to the Amazon ECR repository. Make sure to include :latest at the end of the repository URI.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. **Create an execution role** for the function, if you don’t already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include :latest at the end of the URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \n  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \n  --role arn:aws:iam::111122223333:role/lambda-ex
```

**Note**

You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```json
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.
To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the `update-function-code` command to deploy the image to the Lambda function.
AWS Lambda context object in Ruby

When Lambda runs your function, it passes a context object to the `handler`. This object provides methods and properties that provide information about the invocation, function, and execution environment.

**Context methods**

- `get_remaining_time_in_millis` – Returns the number of milliseconds left before the execution times out.

**Context properties**

- `function_name` – The name of the Lambda function.
- `function_version` – The version of the function.
- `invoked_function_arn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memory_limit_in_mb` – The amount of memory that's allocated for the function.
- `aws_request_id` – The identifier of the invocation request.
- `log_group_name` – The log group for the function.
- `log_stream_name` – The log stream for the function instance.
- `deadline_ms` – The date that the execution times out, in Unix time milliseconds.
- `identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
- `client_context` – (mobile apps) Client context that's provided to Lambda by the client application.
AWS Lambda function logging in Ruby

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see Using Amazon CloudWatch logs with AWS Lambda.

This page describes how to produce log output from your Lambda function's code, or access logs using the AWS Command Line Interface, the Lambda console, or the CloudWatch console.

Sections

- Creating a function that returns logs
- Using the Lambda console
- Using the CloudWatch console
- Using the AWS Command Line Interface (AWS CLI)
- Deleting logs
- Logger library

Creating a function that returns logs

To output logs from your function code, you can use `puts` statements, or any logging library that writes to `stdout` or `stderr`. The following example logs the values of environment variables and the event object.

Example `lambda_function.rb`

```ruby
# lambda_function.rb

def handler(event:, context:)
    puts "## ENVIRONMENT VARIABLES"
    puts ENV.to_a
    puts "## EVENT"
    puts event.to_a
end
```

Example log format

```plaintext
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
        'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c',
        'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Duration: 15.74 ms Billed
  Duration: 16 ms Memory Size: 128 MB Max Memory Used: 56 MB Init Duration: 130.49 ms
  XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1 SegmentId: Ø7f5xmpl2d1f6f85
  Sampled: true
```

The Ruby runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details.

**REPORT line data fields**

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TraceId** – For traced requests, the [AWS X-Ray trace ID](https://aws.amazon.com/xray).
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

For more detailed logs, use the section called “Logger library”.

**Using the Lambda console**

You can use the Lambda console to view log output after you invoke a Lambda function.
If your code can be tested from the embedded Code editor, you will find logs in the execution results. When you use the console test feature to invoke a function, you'll find Log output in the Details section.

**Using the CloudWatch console**

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

**To view logs on the CloudWatch console**

1. Open the Log groups page on the CloudWatch console.
2. Choose the log group for your function (/aws/lambda/your-function-name).
3. Choose a log stream.

Each log stream corresponds to an instance of your function. A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see [Error processor sample application for AWS Lambda](#).

**Using the AWS Command Line Interface (AWS CLI)**

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- AWS Command Line Interface (AWS CLI) version 2
- AWS CLI – Quick configuration with `aws configure`

You can use the AWS CLI to retrieve logs for an invocation using the `--log-type` command option. The response contains a LogResult field that contains up to 4 KB of base64-encoded logs from the invocation.

**Example retrieve a log ID**

The following example shows how to retrieve a log ID from the LogResult field for a function named my-function.
aws lambda invoke --function-name my-function out --log-type Tail

You should see the following output:

```
{
   "StatusCode": 200,
   "LogResult": "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTEzNTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...", 
   "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --decode
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the AWS Command Line Interface User Guide for Version 2.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", ",_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Duration: 79.67 ms Billed
Duration: 80 ms Memory Size: 128 MB Max Memory Used: 73 MB
```

The base64 utility is available on Linux, macOS, and Ubuntu on Windows. macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for
the logs to become available. The output includes the response from Lambda and the output from
the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default
setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more
information, see `AWS CLI supported global command line options` in the `AWS Command Line

```bash
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '"key": "value"'}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-name stream1 --limit 5
```

**Example macOS and Linux (only)**

In the same command prompt, macOS and Linux users may need to run the following command to
ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

**Example retrieve the last five log events**

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```json
{
   "StatusCode": 200,
   "ExecutedVersion": "$LATEST"
}
```
"events": [
    {
        "timestamp": 1559763003171,
        "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST\n",
        "ingestionTime": 1559763003309
    },
    {
        "timestamp": 1559763003173,
        "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES:\r{\r  "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",\r ...",
        "ingestionTime": 1559763018353
    },
    {
        "timestamp": 1559763003173,
        "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  "key": "value"\r}\n",
        "ingestionTime": 1559763018353
    },
    {
        "timestamp": 1559763003218,
        "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
        "ingestionTime": 1559763018353
    },
    {
        "timestamp": 1559763003218,
        "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75 MB\t\n",
        "ingestionTime": 1559763018353
    }
],
"nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
"nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"

## Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.
The Ruby [logger library](https://aws.amazon.com/about-aws/what-is-aws/languages/python-sdk/) returns streamlined logs that are easily read. Use the logger utility to output detailed information, messages, and errors codes related to your function.

```ruby
# lambda_function.rb

require 'logger'

def handler(event:, context:)
  logger = Logger.new($stdout)
  logger.info('## ENVIRONMENT VARIABLES')
  logger.info(ENV.to_a)
  logger.info('## EVENT')
  logger.info(event)
  event.to_a
end
```

The output from `logger` includes the log level, timestamp, and request ID.

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO] 2020-01-31T22:12:58.534Z 1c8df7d3-xmpl-46da-9778-518e6eca8125 ## ENVIRONMENT VARIABLES
[INFO] 2020-01-31T22:12:58.535Z 1c8df7d3-xmpl-46da-9778-518e6eca8125 {'key': 'value'}
END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Duration: 2.75 ms Billed Duration: 3 ms Memory Size: 128 MB Max Memory Used: 56 MB Init Duration: 113.51 ms XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370 SegmentId: 073xmpl3e442861 Sampled: true
```
AWS Lambda function errors in Ruby

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

This page describes how to view Lambda function invocation errors for the Ruby runtime using the Lambda console and the AWS CLI.

Sections

- Syntax
- How it works
- Using the Lambda console
- Using the AWS Command Line Interface (AWS CLI)
- Error handling in other AWS services
- Sample applications
- What's next?

Syntax

Example function.rb

```ruby
def handler(event:, context:)
  puts "Processing event..."
  [1, 2, 3].first("two")
  "Success"
end
```

This code results in a type error. Lambda catches the error and generates a JSON document with fields for the error message, the type, and the stack trace.

```
{
  "errorMessage": "no implicit conversion of String into Integer",
  "errorType": "Function<TypeError>",
  "stackTrace": [
    "/var/task/function.rb:3:in `first`
  
  
  ]
```

Errors
How it works

When you invoke a Lambda function, Lambda receives the invocation request and validates the permissions in your execution role, verifies that the event document is a valid JSON document, and checks parameter values.

If the request passes validation, Lambda sends the request to a function instance. The Lambda runtime environment converts the event document into an object, and passes it to your function handler.

If Lambda encounters an error, it returns an exception type, message, and HTTP status code that indicates the cause of the error. The client or service that invoked the Lambda function can handle the error programatically, or pass it along to an end user. The correct error handling behavior depends on the type of application, the audience, and the source of the error.

The following list describes the range of status codes you can receive from Lambda.

2xx

A 2xx series error with a X-Amz-Function-Error header in the response indicates a Lambda runtime or function error. A 2xx series status code indicates that Lambda accepted the request, but instead of an error code, Lambda indicates the error by including the X-Amz-Function-Error header in the response.

4xx

A 4xx series error indicates an error that the invoking client or service can fix by modifying the request, requesting permission, or by retrying the request. 4xx series errors other than 429 generally indicate an error with the request.

5xx

A 5xx series error indicates an issue with Lambda, or an issue with the function's configuration or resources. 5xx series errors can indicate a temporary condition that can be resolved without any action by the user. These issues can't be addressed by the invoking client or service, but a Lambda function's owner may be able to fix the issue.

For a complete list of invocation errors, see InvokeFunction errors.
Using the Lambda console

You can invoke your function on the Lambda console by configuring a test event and viewing the output. The output is captured in the function's execution logs and, when active tracing is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the Functions page of the Lambda console.
2. Choose the function to test, and choose Test.
3. Under Test event, select New event.
4. Select a Template.
5. For Name, enter a name for the test. In the text entry box, enter the JSON test event.
6. Choose Save changes.
7. Choose Test.

The Lambda console invokes your function synchronously and displays the result. To see the response, logs, and other information, expand the Details section.

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- AWS Command Line Interface (AWS CLI) version 2
- AWS CLI – Quick configuration with aws configure

When you invoke a Lambda function in the AWS CLI, the AWS CLI splits the response into two documents. The AWS CLI response is displayed in your command prompt. If an error has occurred, the response contains a FunctionError field. The invocation response or error returned by the function is written to an output file. For example, output.json or output.txt.

The following invoke command example demonstrates how to invoke a function and write the invocation response to an output.txt file.

```bash
code
aws lambda invoke
```
The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the AWS Command Line Interface User Guide for Version 2.

You should see the AWS CLI response in your command prompt:

```
{
   "StatusCode": 200,
   "FunctionError": "Unhandled",
   "ExecutedVersion": "$LATEST"
}
```

You should see the function invocation response in the `output.txt` file. In the same command prompt, you can also view the output in your command prompt using:

```
cat output.txt
```

You should see the invocation response in your command prompt.

```
{"errorMessage":"no implicit conversion of String into Integer","errorType":"Function<TypeError>","stackTrace":[/var/task/function.rb:3:in `first',/var/task/function.rb:3:in `handler']}
```

## Error handling in other AWS services

When another AWS service invokes your function, the service chooses the invocation type and retry behavior. AWS services can invoke your function on a schedule, in response to a lifecycle event on a resource, or to serve a request from a user. Some services invoke functions asynchronously and let Lambda handle errors, while others retry or pass errors back to the user.

For example, API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502 error code. To customize the error response, you must catch errors in your code and format a response in the required format.
We recommend using AWS X-Ray to determine the source of an error and its cause. X-Ray allows you to find out which component encountered an error, and see details about the errors. The following example shows a function error that resulted in a 502 response from API Gateway.

For more information, see Instrumenting Ruby code in AWS Lambda.

Sample applications

The following sample code is available for the Ruby runtime.

Sample Lambda applications in Ruby

- **blank-ruby** – A Ruby function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.
- **Ruby Code Samples for AWS Lambda** – Code samples written in Ruby that demonstrate how to interact with AWS Lambda.

What's next?

- Learn how to show logging events for your Lambda function on the the section called “Logging” page.
Instrumenting Ruby code in AWS Lambda

Lambda integrates with AWS X-Ray to enable you to trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, from the frontend API to storage and database on the backend. By simply adding the X-Ray SDK library to your build configuration, you can record errors and latency for any call that your function makes to an AWS service.

After you've configured active tracing, you can observe specific requests through your application. The X-Ray service graph shows information about your application and all its components. The following example from the error processor sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.

To toggle active tracing on your Lambda function with the console, follow these steps:

**To turn on active tracing**

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose Configuration and then choose Monitoring and operations tools.
4. Choose Edit.
5. Under X-Ray, toggle on Active tracing.
6. Choose Save.
### Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see [AWS X-Ray pricing](#).

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's **execution role**. Otherwise, add the **AWSXRayDaemonWriteAccess** policy to the execution role.

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

#### Note

You cannot configure the X-Ray sampling rate for your functions.

When using active tracing, Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes for the primary function from the **error processor sample application**.

The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of **AWS::Lambda** and the other has origin **AWS::Lambda::Function**.
This example expands the function segment to show its three subsegments:

- **Initialization** – Represents time spent loading your function and running **initialization code**. This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

You can instrument your handler code to record metadata and trace downstream calls. To record detail about calls that your handler makes to other resources and services, use the X-Ray SDK for Ruby. To get the SDK, add the `aws-xray-sdk` package to your application's dependencies.

**Example blank-ruby/function/Gemfile**

```ruby
# Gemfile
source 'https://rubygems.org'

gem 'aws-xray-sdk', '0.11.4'
gem 'aws-sdk-lambda', '1.39.0'
gem 'test-unit', '3.3.5'
```

To instrument AWS SDK clients, require the `aws-xray-sdk/lambda` module after creating a client in initialization code.

**Example blank-ruby/function/lambda_function.rb – Tracing an AWS SDK client**

```ruby
# lambda_function.rb
require 'logger'
```
require 'json'
require 'aws-sdk-lambda'
$client = Aws::Lambda::Client.new()
$client.get_account_settings()

require 'aws-xray-sdk/lambda'
def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  ...

When using active tracing, Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes for the primary function from the error processor sample application.

The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named my-function, but one has an origin of AWS::Lambda and the other has origin AWS::Lambda::Function.
This example expands the function segment to show its three subsegments:

- **Initialization** – Represents time spent loading your function and running initialization code. This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see *The X-Ray SDK for Ruby* in the AWS X-Ray Developer Guide.

**Sections**

- Enabling active tracing with the Lambda API
- Enabling active tracing with AWS CloudFormation
- Storing runtime dependencies in a layer

**Enabling active tracing with the Lambda API**

To manage tracing configuration with the AWS CLI or AWS SDK, use the following API operations:

- **UpdateFunctionConfiguration**
- **GetFunctionConfiguration**
- **CreateFunction**

The following example AWS CLI command enables active tracing on a function named `my-function`.

```
aws lambda update-function-configuration --function-name my-function \ --tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.
Enabling active tracing with AWS CloudFormation

To activate tracing on an AWS::Lambda::Function resource in an AWS CloudFormation template, use the TracingConfig property.

Example function-inline.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

For an AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function resource, use the Tracing property.

Example template.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Storing runtime dependencies in a layer

If you use the X-Ray SDK to instrument AWS SDK clients your function code, your deployment package can become quite large. To avoid uploading runtime dependencies every time you update your function code, package the X-Ray SDK in a Lambda layer.

The following example shows an AWS::Serverless::LayerVersion resource that stores X-Ray SDK for Ruby.

Example template.yml – Dependencies layer

```
Resources:
  function:
    Type: AWS::Serverless::Function
```
With this configuration, you update the library layer only if you change your runtime dependencies. Since the function deployment package contains only your code, this can help reduce upload times.

Creating a layer for dependencies requires build changes to generate the layer archive prior to deployment. For a working example, see the blank-ruby sample application.
Building Lambda functions with Java

You can run Java code in AWS Lambda. Lambda provides runtimes for Java that run your code to process events. Your code runs in an Amazon Linux environment that includes AWS credentials from an AWS Identity and Access Management (IAM) role that you manage.

Lambda supports the following Java runtimes.

Java

<table>
<thead>
<tr>
<th>Name</th>
<th>Identifier</th>
<th>Operating system</th>
<th>Deprecation date</th>
<th>Block function create</th>
<th>Block function update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java 21</td>
<td>java21</td>
<td>Amazon Linux 2023</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java 17</td>
<td>java17</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java 11</td>
<td>java11</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java 8</td>
<td>java8.al2</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java 8</td>
<td>java8</td>
<td>Amazon Linux</td>
<td>Jan 8, 2023</td>
<td>Feb 8, 2024</td>
<td>Mar 12, 2024</td>
</tr>
</tbody>
</table>

Lambda provides the following libraries for Java functions:

- **com.amazonaws:aws-lambda-java-core** (required) – Defines handler method interfaces and the context object that the runtime passes to the handler. If you define your own input types, this is the only library that you need.
- **com.amazonaws:aws-lambda-java-events** – Input types for events from services that invoke Lambda functions.
- **com.amazonaws:aws-lambda-java-log4j2** – An appender library for Apache Log4j 2 that you can use to add the request ID for the current invocation to your function logs.
• **AWS SDK for Java 2.0** – The official AWS SDK for the Java programming language.

⚠️ **Important**

Don't use private components of the JDK API, such as private fields, methods, or classes. Non-public API components can change or be removed in any update, causing your application to break.

---

**To create a Java function**

1. Open the [Lambda console](https://console.aws.amazon.com/lambda/home).
2. Choose [Create function](https://console.aws.amazon.com/lambda/home).
3. Configure the following settings:

   - **Function name**: Enter a name for the function.
5. To configure a test event, choose [Test](https://console.aws.amazon.com/lambda/home).
6. For **Event name**, enter `test`.
7. Choose [Save changes](https://console.aws.amazon.com/lambda/home).
8. To invoke the function, choose [Test](https://console.aws.amazon.com/lambda/home).

The console creates a Lambda function with a handler class named **Hello**. Since Java is a compiled language, you can't view or edit the source code in the Lambda console, but you can modify its configuration, invoke it, and configure triggers.

⚠️ **Note**

To get started with application development in your local environment, deploy one of the sample applications available in this guide's GitHub repository.

---

The **Hello** class has a function named `handleRequest` that takes an event object and a context object. This is the **handler function** that Lambda calls when the function is invoked. The Java
function runtime gets invocation events from Lambda and passes them to the handler. In the function configuration, the handler value is `example.Hello::handleRequest`.

To update the function's code, you create a deployment package, which is a .zip file archive that contains your function code. As your function development progresses, you will want to store your function code in source control, add libraries, and automate deployments. Start by creating a deployment package and updating your code at the command line.

The function runtime passes a context object to the handler, in addition to the invocation event. The `context object` contains additional information about the invocation, the function, and the execution environment. More information is available from environment variables.

Your Lambda function comes with a CloudWatch Logs log group. The function runtime sends details about each invocation to CloudWatch Logs. It relays any logs that your function outputs during invocation. If your function returns an error, Lambda formats the error and returns it to the invoker.

**Topics**

- [AWS Lambda function handler in Java](#)
- [Deploy Java Lambda functions with .zip or JAR file archives](#)
- [Deploy Java Lambda functions with container images](#)
- [Java Lambda function customization settings](#)
- [AWS Lambda context object in Java](#)
- [AWS Lambda function logging in Java](#)
- [AWS Lambda function errors in Java](#)
- [Instrumenting Java code in AWS Lambda](#)
- [Creating a deployment package using Eclipse](#)
- [Java sample applications for AWS Lambda](#)
AWS Lambda function handler in Java

The Lambda function handler is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

The GitHub repo for this guide provides easy-to-deploy sample applications that demonstrate a variety of handler types. For details, see the end of this topic.

Sections

• Example handler: Java 17 runtimes
• Example handler: Java 11 runtimes and below
• Initialization code
• Choosing input and output types
• Handler interfaces
• Sample handler code

Example handler: Java 17 runtimes

In the following Java 17 example, a class named HandlerIntegerJava17 defines a handler method named handleRequest. The handler method takes in the following inputs:

• An IntegerRecord, which is a custom Java record that represents event data. In this example, we define IntegerRecord as follows:

```java
record IntegerRecord(int x, int y, String message) {
}
```

• A context object, which provides methods and properties that provide information about the invocation, function, and execution environment.

Suppose we want to write a function that logs the message from the input IntegerRecord, and returns the sum of x and y. The following is the function code:

Example HandlerIntegerJava17.java

```java
package example;
```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;

// Handler value: example.HandlerInteger
public class HandlerIntegerJava17 implements RequestHandler<IntegerRecord, Integer>{

    @Override
    /*
     * Takes in an InputRecord, which contains two integers and a String.
     * Logs the String, then returns the sum of the two Integers.
     */
    public Integer handleRequest(IntegerRecord event, Context context) {
        LambdaLogger logger = context.getLogger();
        logger.log("String found: " + event.message());
        return event.x() + event.y();
    }
}

record IntegerRecord(int x, int y, String message) {
}

You specify which method you want Lambda to invoke by setting the handler parameter on your function's configuration. You can express the handler in the following formats:

- **package.Class::method** – Full format. For example: example.Handler::handleRequest.
- **package.Class** – Abbreviated format for classes that implement a **handler interface**. For example: example.Handler.

When Lambda invokes your handler, the **Lambda runtime** receives an event as a JSON-formatted string and converts it into an object. For the previous example, a sample event might look like the following:

**Example event.json**

```json
{
    "x": 1,
    "y": 20,
    "message": "Hello World!"
}
```
You can save this file and test your function locally with the following AWS Command Line Interface (CLI) command:

```
aws lambda invoke --function-name function_name --payload file://event.json out.json
```

**Example handler: Java 11 runtimes and below**

Lambda supports records in Java 17 and later runtimes. In all Java runtimes, you can use a class to represent event data. The following example takes a list of integers and a context object as input, and returns the sum of all integers in the list.

**Example** [Handler.java](#)

In the following example, a class named `Handler` defines a handler method named `handleRequest`. The handler method takes an event and context object as input and returns a string.

**Example** [HandlerList.java](#)

```java
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import java.util.List;
// Handler value: example.HandlerList
public class HandlerList implements RequestHandler<List<Integer>, Integer>{
   @Override
   /*
    * Takes a list of Integers and returns its sum.
    */
   public Integer handleRequest(List<Integer> event, Context context) {
      LambdaLogger logger = context.getLogger();
      logger.log("EVENT TYPE: " + event.getClass().toString());
      return event.stream().mapToInt(Integer::intValue).sum();
   }
```

For more examples, see Sample handler code.

Initialization code

Lambda runs your static code and the class constructor during the initialization phase before invoking your function for the first time. Resources created during initialization stay in memory between invocations and can be reused by the handler thousands of times. Thus, you can add initialization code outside of your main handler method to save compute time and reuse resources across multiple invocations.

In the following example, the client initialization code is outside the main handler method. The runtime initializes the client before the function serves its first event. Subsequent events are much faster because Lambda doesn't need to initialize the client again.

Example Handler.java

```java
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import java.util.Map;
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.GetAccountSettingsResponse;
import software.amazon.awssdk.services.lambda.model.LambdaException;

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, String> {
    private static final LambdaClient lambdaClient = LambdaClient.builder().build();

    @Override
    public String handleRequest(Map<String,String> event, Context context) {
        LambdaLogger logger = context.getLogger();
        logger.log("Handler invoked");
    }
}
```
Choosing input and output types

You specify the type of object that the event maps to in the handler method's signature. In the preceding example, the Java runtime deserializes the event into a type that implements the Map<String,String> interface. String-to-string maps work for flat events like the following:

Example **Event.json** – Weather data

```json
{
  "temperatureK": 281,
  "windKmh": -3,
  "humidityPct": 0.55,
  "pressureHPa": 1020
}
```

However, the value of each field must be a string or number. If the event includes a field that has an object as a value, the runtime can't deserialize it and returns an error.

Choose an input type that works with the event data that your function processes. You can use a basic type, a generic type, or a well-defined type.

**Input types**

- **Integer, Long, Double, etc.** – The event is a number with no additional formatting—for example, 3.5. The runtime converts the value into an object of the specified type.
- **String** – The event is a JSON string, including quotes—for example, "My string.". The runtime converts the value (without quotes) into a String object.
- **Type, Map<String, Type>** etc. – The event is a JSON object. The runtime deserializes it into an object of the specified type or interface.
• List<Integer>, List<String>, List<Object>, etc. – The event is a JSON array. The runtime deserializes it into an object of the specified type or interface.

• InputStream – The event is any JSON type. The runtime passes a byte stream of the document to the handler without modification. You deserialize the input and write output to an output stream.

• Library type – For events sent by AWS services, use the types in the [aws-lambda-java-events](#) library.

If you define your own input type, it should be a deserializable, mutable plain old Java object (POJO), with a default constructor and properties for each field in the event. Keys in the event that don't map to a property as well as properties that aren't included in the event are dropped without error.

The output type can be an object or `void`. The runtime serializes return values into text. If the output is an object with fields, the runtime serializes it into a JSON document. If it's a type that wraps a primitive value, the runtime returns a text representation of that value.

**Handler interfaces**

The [aws-lambda-java-core](#) library defines two interfaces for handler methods. Use the provided interfaces to simplify handler configuration and validate the handler method signature at compile time.

• `com.amazonaws.services.lambda.runtime.RequestHandler`
• `com.amazonaws.services.lambda.runtime.RequestStreamHandler`

The `RequestHandler` interface is a generic type that takes two parameters: the input type and the output type. Both types must be objects. When you use this interface, the Java runtime deserializes the event into an object with the input type, and serializes the output into text. Use this interface when the built-in serialization works with your input and output types.

**Example Handler.java – Handler interface**

```java
// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, String>{
   @Override
   public String handleRequest(Map<String,String> event, Context context)
```
To use your own serialization, implement the `RequestStreamHandler` interface. With this interface, Lambda passes your handler an input stream and output stream. The handler reads bytes from the input stream, writes to the output stream, and returns void.

The following example uses buffered reader and writer types to work with the input and output streams.

Example **HandlerStream.java**

```java
import com.amazonaws.services.lambda.runtime.Context
import com.amazonaws.services.lambda.runtime.LambdaLogger
import com.amazonaws.services.lambda.runtime.RequestStreamHandler

// Handler value: example.HandlerStream
public class HandlerStream implements RequestStreamHandler {
    @Override
    /**
     * Takes an InputStream and an OutputStream. Reads from the InputStream,
     * and copies all characters to the OutputStream.
     */
    public void handleRequest(InputStream inputStream, OutputStream outputStream, Context context) throws IOException {
        LambdaLogger logger = context.getLogger();
        BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream, Charset.forName("US-ASCII")));
        PrintWriter writer = new PrintWriter(new BufferedWriter(new OutputStreamWriter(outputStream, Charset.forName("US-ASCII"))));
        int nextChar;
        try {
            while ((nextChar = reader.read()) != -1) {
                outputStream.write(nextChar);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            reader.close();
            String finalString = writer.toString();
            logger.log("Final string result: " + finalString);
            writer.close();
        }
    }
}
```
Sample handler code

The GitHub repository for this guide includes sample applications that demonstrate the use of various handler types and interfaces. Each sample application includes scripts for easy deployment and cleanup, an AWS SAM template, and supporting resources.

Sample Lambda applications in Java

- **java17-examples** – A Java function that demonstrates how to use a Java record to represent an input event data object.
- **java-basic** – A collection of minimal Java functions with unit tests and variable logging configuration.
- **java-events** – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the [aws-lambda-java-events](https://aws.amazon.com) library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- **s3-java** – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- **Use API Gateway to invoke a Lambda function** – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.

The java-events and s3-java applications take an AWS service event as input and return a string. The java-basic application includes several types of handlers:

- **Handler.java** – Takes a Map<String,String> as input.
- **HandlerInteger.java** – Takes an Integer as input.
- **HandlerList.java** – Takes a List<Integer> as input.
- **HandlerStream.java** – Takes an InputStream and OutputStream as input.
- **HandlerString.java** – Takes a String as input.
- **HandlerWeatherData.java** – Takes a custom type as input.

To test different handler types, just change the handler value in the AWS SAM template. For detailed instructions, see the sample application’s readme file.
Deploy Java Lambda functions with .zip or JAR file archives

Your AWS Lambda function’s code consists of scripts or compiled programs and their dependencies. You use a deployment package to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives.

This page describes how to create your deployment package as a .zip file or Jar file, and then use the deployment package to deploy your function code to AWS Lambda using the AWS Command Line Interface (AWS CLI).

Sections
- Prerequisites
- Tools and libraries
- Building a deployment package with Gradle
- Creating a Java layer for your dependencies
- Building a deployment package with Maven
- Uploading a deployment package with the Lambda console
- Uploading a deployment package with the AWS CLI
- Uploading a deployment package with AWS SAM

Prerequisites

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- AWS Command Line Interface (AWS CLI) version 2
- AWS CLI – Quick configuration with aws configure

Tools and libraries

Lambda provides the following libraries for Java functions:

- com.amazonaws.aws-lambda-java-core (required) – Defines handler method interfaces and the context object that the runtime passes to the handler. If you define your own input types, this is the only library that you need.
• **com.amazonaws:aws-lambda-java-events** – Input types for events from services that invoke Lambda functions.

• **com.amazonaws:aws-lambda-java-log4j2** – An appender library for Apache Log4j 2 that you can use to add the request ID for the current invocation to your function logs.

• **AWS SDK for Java 2.0** – The official AWS SDK for the Java programming language.

These libraries are available through Maven Central Repository. Add them to your build definition as follows:

**Gradle**

```
dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
}
```

**Maven**

```
<dependencies>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-lambda-java-core</artifactId>
        <version>1.2.2</version>
    </dependency>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-lambda-java-events</artifactId>
        <version>3.11.1</version>
    </dependency>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-lambda-java-log4j2</artifactId>
        <version>1.5.1</version>
    </dependency>
</dependencies>
```

To create a deployment package, compile your function code and dependencies into a single .zip file or Java Archive (JAR) file. For Gradle, use the Zip build type. For Apache Maven, use the Maven

Tools and libraries

685
**Shade plugin.** To upload your deployment package, use the Lambda console, the Lambda API, or AWS Serverless Application Model (AWS SAM).

---

**Note**

To keep your deployment package size small, package your function's dependencies in layers. Layers enable you to manage your dependencies independently, can be used by multiple functions, and can be shared with other accounts. For more information, see [Lambda layers](#).

---

### Building a deployment package with Gradle

To create a deployment package with your function's code and dependencies in Gradle, use the Zip build type. Here's an example from a [complete sample build.gradle file](#):

```groovy
Example build.gradle – Build task

task buildZip(type: Zip) {
    into('lib') {
        from(jar)
        from(configurations.runtimeClasspath)
    }
}
```

This build configuration produces a deployment package in the build/distributions directory. Within the `into('lib')` statement, the `jar` task assembles a jar archive containing your main classes into a folder named `lib`. Additionally, the `configurations.runtimeClassPath` task copies dependency libraries from the build's classpath into the same `lib` folder.

**Example build.gradle – Dependencies**

```groovy
dependencies {
    ... 
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    implementation 'org.apache.logging.log4j:log4j-api:2.17.1'
    implementation 'org.apache.logging.log4j:log4j-core:2.17.1'
    runtimeOnly 'org.apache.logging.log4j:log4j-slf4j18-impl:2.17.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
}
```
Lambda loads JAR files in Unicode alphabetical order. If multiple JAR files in the `lib` directory contain the same class, the first one is used. You can use the following shell script to identify duplicate classes:

```
Example test-zip.sh

mkdir -p expanded
unzip path/to/my/function.zip -d expanded
find ./expanded/lib -name '*.jar' | xargs -n1 zipinfo -1 | grep '.*.class' | sort |
uniq -c | sort
```

Creating a Java layer for your dependencies

**Note**

Using layers with functions in a compiled language like Java may not provide the same amount of benefit as with an interpreted language like Python. Since Java is a compiled language, your functions still have to manually load any shared assemblies into memory during the init phase, which can increase cold start times. Instead, we recommend including any shared code at compile time to take advantage of any built-in compiler optimizations.

The instructions in this section show you how to include your dependencies in a layer. For instructions on how to include your dependencies in your deployment package, see the section called “Building a deployment package with Gradle” or the section called “Building a deployment package with Maven”.

When you add a layer to a function, Lambda loads the layer content into the `/opt` directory of that execution environment. For each Lambda runtime, the PATH variable already includes specific folder paths within the `/opt` directory. To ensure that your layer content gets picked up by the PATH variable, include the content in the following folder paths:

- `java/lib (CLASSPATH)`

For example, your layer `.zip` file structure might look like the following:
In addition, Lambda automatically detects any libraries in the `/opt/lib` directory, and any binaries in the `/opt/bin` directory. To ensure that Lambda properly finds your layer content, you can also create a layer with the following structure:

```
custom-layer.zip
# lib
  | lib_1
  | lib_2
# bin
  | bin_1
  | bin_2
```

After you package your layer, see the section called “Creating and deleting layers” and the section called “Adding layers” to complete your layer setup.

**Building a deployment package with Maven**

To build a deployment package with Maven, use the [Maven Shade plugin](https://maven.apache.org/plugins/maven-shade-plugin/

Example pom.xml – Plugin configuration

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>3.2.2</version>
  <configuration>
    <createDependencyReducedPom>false</createDependencyReducedPom>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```
To build the deployment package, use the `mvn package` command.

This command generates a JAR file in the `target` directory.

**Note**

If you're working with a multi-release JAR (MRJAR), you must include the MRJAR (i.e. the shaded JAR produced by the Maven Shade plugin) in the `lib` directory and zip it before uploading your deployment package to Lambda. Otherwise, Lambda may not properly unpack your JAR file, causing your MANIFEST.MF file to be ignored.

If you use the appender library (aws-lambda-java-log4j2), you must also configure a transformer for the Maven Shade plugin. The transformer library combines versions of a cache file that appear in both the appender library and in Log4j.

**Example pom.xml – Plugin configuration with Log4j 2 appender**

```xml
<plugin>
```
<groupId>org.apache.maven.plugins</groupId>  
<artifactId>maven-shade-plugin</artifactId>  
.getVersion>3.2.2</version>  
<configuration>  
  <createDependencyReducedPom>false</createDependencyReducedPom>  
</configuration>  
<executions>  
  <execution>  
    <phase>package</phase>  
    <goals>  
      <goal>shade</goal>  
    </goals>  
    <configuration>  
      <transformers>  
        <transformer  
          implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCacheFileTransformer">  
        </transformer>  
      </transformers>  
    </configuration>  
  </execution>  
</executions>  
<dependencies>  
  <dependency>  
    <groupId>com.github.edwgiz</groupId>  
    <artifactId>maven-shade-plugin.log4j2-cachefile-transformer</artifactId>  
    <version>2.13.0</version>  
  </dependency>  
</dependencies>  
</plugin>

### Uploading a deployment package with the Lambda console

To create a new function, you must first create the function in the console, then upload your .zip or JAR file. To update an existing function, open the page for your function, then follow the same procedure to add your updated .zip or JAR file.

If your deployment package file is less than 50MB, you can create or update a function by uploading the file directly from your local machine. For .zip or JAR files greater than 50MB, you must upload your package to an Amazon S3 bucket first. For instructions on how to upload a file to an Amazon S3 bucket using the AWS Management Console, see [Getting started with Amazon S3](#). To upload files using the AWS CLI, see [Move objects](#) in the [AWS CLI User Guide](#).
To create a new function (console)

1. Open the Functions page of the Lambda console and choose Create Function.
2. Choose Author from scratch.
3. Under Basic information, do the following:
   a. For Function name, enter the name for your function.
   b. For Runtime, select the runtime you want to use.
   c. (Optional) For Architecture, choose the instruction set architecture for your function. The default architecture is x86_64. Ensure that the .zip deployment package for your function is compatible with the instruction set architecture you select.
4. (Optional) Under Permissions, expand Change default execution role. You can create a new Execution role or use an existing one.
5. Choose Create function. Lambda creates a basic 'Hello world' function using your chosen runtime.

To upload a .zip or JAR archive from your local machine (console)

1. In the Functions page of the Lambda console, choose the function you want to upload the .zip or JAR file for.
2. Select the Code tab.
3. In the Code source pane, choose Upload from.
4. Choose .zip or .jar file.
5. To upload the .zip or JAR file, do the following:
   a. Select Upload, then select your .zip or JAR file in the file chooser.
   b. Choose Open.
   c. Choose Save.
To upload a .zip or JAR archive from an Amazon S3 bucket (console)

1. In the Functions page of the Lambda console, choose the function you want to upload a new .zip or JAR file for.
2. Select the Code tab.
3. In the Code source pane, choose Upload from.
4. Choose Amazon S3 location.
5. Paste the Amazon S3 link URL of your .zip file and choose Save.

Uploading a deployment package with the AWS CLI

You can use the AWS CLI to create a new function or to update an existing one using a .zip or JAR file. Use the create-function and update-function-code commands to deploy your .zip or JAR package. If your file is smaller than 50MB, you can upload the package from a file location on your local build machine. For larger files, you must upload your .zip or JAR package from an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the AWS CLI User Guide.

Note

If you upload your .zip or JAR file from an Amazon S3 bucket using the AWS CLI, the bucket must be located in the same AWS Region as your function.

To create a new function using a .zip or JAR file with the AWS CLI, you must specify the following:

- The name of your function (--function-name)
- Your function’s runtime (--runtime)
- The Amazon Resource Name (ARN) of your function’s execution role (--role)
- The name of the handler method in your function code (--handler)

You must also specify the location of your .zip or JAR file. If your .zip or JAR file is located in a folder on your local build machine, use the --zip-file option to specify the file path, as shown in the following example command.

```
aws lambda create-function --function-name myFunction
```
To specify the location of .zip file in an Amazon S3 bucket, use the `--code` option as shown in the following example command. You only need to use the `S3ObjectVersion` parameter for versioned objects.

```
aws lambda create-function --function-name myFunction \
--runtime java21 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
--code S3Bucket=myBucketName,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

To update an existing function using the CLI, you specify the the name of your function using the `--function-name` parameter. You must also specify the location of the .zip file you want to use to update your function code. If your .zip file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the `--s3-bucket` and `--s3-key` options as shown in the following example command. You only need to use the `--s3-object-version` parameter for versioned objects.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket myBucketName --s3-key myFileName.zip --s3-object-version myObject Version
```

### Uploading a deployment package with AWS SAM

You can use AWS SAM to automate deployments of your function code, configuration, and dependencies. AWS SAM is an extension of AWS CloudFormation that provides a simplified syntax for defining serverless applications. The following example template defines a function with a deployment package in the `build/distributions` directory that Gradle uses:

**Example template.yml**

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
```
Description: An AWS Lambda application that calls the Lambda API.

Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/java-basic.zip
      Handler: example.Handler
      Runtime: java21
      Description: Java function
      MemorySize: 512
      Timeout: 10
      # Function's execution role
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
        - AWSServerlessRole
      Tracing: Active

To create the function, use the package and deploy commands. These commands are customizations to the AWS CLI. They wrap other commands to upload the deployment package to Amazon S3, rewrite the template with the object URI, and update the function's code.

The following example script runs a Gradle build and uploads the deployment package that it creates. It creates an AWS CloudFormation stack the first time you run it. If the stack already exists, the script updates it.

Example deploy.sh

```bash
#!/bin/bash
set -eo pipefail
aws cloudformation package --template-file template.yml --s3-bucket MY_BUCKET --output-template-file out.yml
aws cloudformation deploy --template-file out.yml --stack-name java-basic --capabilities CAPABILITY_NAMED_IAM
```

For a complete working example, see the following sample applications:

Sample Lambda applications in Java

- [java17-examples](#) – A Java function that demonstrates how to use a Java record to represent an input event data object.
• **java-basic** – A collection of minimal Java functions with unit tests and variable logging configuration.

• **java-events** – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the [aws-lambda-java-events](https://aws-lambda-java-events) library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.

• **s3-java** – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.

• **Use API Gateway to invoke a Lambda function** – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.
Deploy Java Lambda functions with container images

There are three ways to build a container image for a Java Lambda function:

- **Using an AWS base image for Java**

  The [AWS base images](#) are preloaded with a language runtime, a runtime interface client to manage the interaction between Lambda and your function code, and a runtime interface emulator for local testing.

- **Using an AWS OS-only base image**

  [AWS OS-only base images](#) contain an Amazon Linux distribution and the [runtime interface emulator](#). These images are commonly used to create container images for compiled languages, such as [Go](#) and [Rust](#), and for a language or language version that Lambda doesn't provide a base image for, such as Node.js 19. You can also use OS-only base images to implement a [custom runtime](#). To make the image compatible with Lambda, you must include the [runtime interface client for Java](#) in the image.

- **Using a non-AWS base image**

  You can use an alternative base image from another container registry, such as Alpine Linux or Debian. You can also use a custom image created by your organization. To make the image compatible with Lambda, you must include the [runtime interface client for Java](#) in the image.

**Tip**

To reduce the time it takes for Lambda container functions to become active, see [Use multi-stage builds](#) in the Docker documentation. To build efficient container images, follow the [Best practices for writing Dockerfiles](#).

This page explains how to build, test, and deploy container images for Lambda.

**Topics**

- [AWS base images for Java](#)
- [Using an AWS base image for Java](#)
- [Using an alternative base image with the runtime interface client](#)
AWS base images for Java

AWS provides the following base images for Java:

<table>
<thead>
<tr>
<th>Tags</th>
<th>Runtime</th>
<th>Operating system</th>
<th>Dockerfile</th>
<th>Deprecation</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>Java 21</td>
<td>Amazon Linux 2023</td>
<td><a href="#">Dockerfile for Java 21 on GitHub</a></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Java 17</td>
<td>Amazon Linux 2</td>
<td><a href="#">Dockerfile for Java 17 on GitHub</a></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Java 11</td>
<td>Amazon Linux 2</td>
<td><a href="#">Dockerfile for Java 11 on GitHub</a></td>
<td></td>
</tr>
<tr>
<td>8.al2</td>
<td>Java 8</td>
<td>Amazon Linux 2</td>
<td><a href="#">Dockerfile for Java 8 on GitHub</a></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Java 8</td>
<td>Amazon Linux</td>
<td><a href="#">Dockerfile for Java 8 on GitHub</a></td>
<td>Jan 8, 2023</td>
</tr>
</tbody>
</table>

Amazon ECR repository: [gallery.ecr.aws/lambda/java](#)

Using an AWS base image for Java

Prerequisites

To complete the steps in this section, you must have the following:

- Java (for example, [Amazon Corretto](#))
- [Docker](#)
- [Apache Maven](#) or [Gradle](#)
- [AWS Command Line Interface (AWS CLI) version 2](#)
Creating an image from a base image

Maven

1. Run the following command to create a Maven project using the archetype for Lambda.
   The following parameters are required:

   - **service** – The AWS service client to use in the Lambda function. For a list of available sources, see [aws-sdk-java-v2/services](https://github.com/aws/aws-sdk-java-v2/services) on GitHub.
   - **region** – The AWS Region where you want to create the Lambda function.
   - **groupId** – The full package namespace of your application.
   - **artifactId** – Your project name. This becomes the name of the directory for your project.

   In Linux and macOS, run this command:

   ```
   mvn -B archetype:generate \
   -DarchetypeGroupId=software.amazon.awssdk \
   -DarchetypeArtifactId=archetype-lambda -Dservice=s3 -Dregion=US_WEST_2 \
   -DgroupId=com.example.myapp \
   -DartifactId=myapp
   ```

   In PowerShell, run this command:

   ```
   mvn -B archetype:generate ` 
   "-DarchetypeGroupId=software.amazon.awssdk" ` 
   "-DarchetypeArtifactId=archetype-lambda" "-Dservice=s3" "-Dregion=US_WEST_2" ` 
   "-DgroupId=com.example.myapp" ` 
   "-DartifactId=myapp"
   ```

   The Maven archetype for Lambda is preconfigured to compile with Java SE 8 and includes a dependency to the AWS SDK for Java. If you create your project with a different archetype or by using another method, you must configure the Java compiler for Maven and declare the SDK as a dependency.

2. Open the `myapp/src/main/java/com/example/myapp` directory, and find the `App.java` file. This is the code for the Lambda function. You can use the provided sample code for testing, or replace it with your own.
3. Navigate back to the project's root directory, and then create a new Dockerfile with the following configuration:

- Set the FROM property to the URI of the base image.
- Set the CMD argument to the Lambda function handler.

**Example Dockerfile**

```dockerfile
FROM public.ecr.aws/lambda/java:21

# Copy function code and runtime dependencies from Maven layout
COPY target/classes ${LAMBDA_TASK_ROOT}
COPY target/dependency/* ${LAMBDA_TASK_ROOT}/lib/

# Set the CMD to your handler (could also be done as a parameter override outside of the Dockerfile)
CMD [ "com.example.myapp.App::handleRequest" ]
```

4. Compile the project and collect the runtime dependencies.

```
mvn compile dependency:copy-dependencies -DincludeScope=runtime
```

5. Build the Docker image with the `docker build` command. The following example names the image `docker-image` and gives it the test tag.

```
docker build --platform linux/amd64 -t docker-image:test .
```

**Note**

The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

**Gradle**

1. Create a directory for the project, and then switch to that directory.
2. Run the following command to have Gradle generate a new Java application project in the example directory in your environment. For **Select build script DSL**, choose **2: Groovy**.

```
gradle init --type java-application
```

3. Open the `/example/app/src/main/java/example` directory, and find the `App.java` file. This is the code for the Lambda function. You can use the following sample code for testing, or replace it with your own.

**Example App.java**

```java
package com.example;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
public class App implements RequestHandler<Object, String> {
    public String handleRequest(Object input, Context context) {
        return "Hello world!";
    }
}
```

4. Open the `build.gradle` file. If you're using the sample function code from the previous step, replace the contents of `build.gradle` with the following. If you're using your own function code, modify your `build.gradle` file as needed.

**Example build.gradle (Groovy DSL)**

```groovy
plugins {
    id 'java'
}
group 'com.example'
version '1.0-SNAPSHOT'
sourceCompatibility = 1.8
repositories {
    mavenCentral()
}
dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.1'
}
```
5. The `gradle init` command from step 2 also generated a dummy test case in the `app/test` directory. For the purposes of this tutorial, skip running tests by deleting the `/test` directory.

6. Build the project.

```
gradle build
```

7. In the project's root directory (`/example`), create a Dockerfile with the following configuration:

- Set the `FROM` property to the [URI of the base image](https://aws.amazon.com/lambda/).
- Use the `COPY` command to copy the function code and runtime dependencies to `{LAMBDA_TASK_ROOT}`.
- Set the `CMD` argument to the Lambda function handler.

**Example Dockerfile**

```
FROM public.ecr.aws/lambda/java:21

# Copy function code and runtime dependencies from Gradle layout
COPY app/build/classes/java/main ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside of the Dockerfile)
CMD [ "com.example.App::handleRequest" ]
```

8. Build the Docker image with the `docker build` command. The following example names the image `docker-image` and gives it the test tag.

```
docker build --platform linux/amd64 -t docker-image:test .
```
Note

The command specifies the --platform linux/amd64 option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the --platform linux/arm64 option instead.

(Optional) Test the image locally

1. Start the Docker image with the docker run command. In this example, docker-image is the image name and test is the tag.

   ```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   This command runs the image as a container and creates a local endpoint at localhost:9000/2015-03-31/functions/function/invocations.

   Note

   If you built the Docker image for the ARM64 instruction set architecture, be sure to use the --platform linux/arm64 option instead of --platform linux/amd64.

2. From a new terminal window, post an event to the local endpoint.

   Linux/macOS

   In Linux and macOS, run the following curl command:

   ```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:
PowerShell

In PowerShell, run the following Invoke-WebRequest command:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{}' -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType "application/json"
```

3. Get the container ID.

```
docker ps
```

4. Use the `docker kill` command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

**Deploying the image**

**To upload the image to Amazon ECR and create the Lambda function**

1. Run the `get-login-password` command to authenticate the Docker CLI to your Amazon ECR registry.

   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace `111122223333` with your AWS account ID.
2. Create a repository in Amazon ECR using the `create-repository` command.

```bash
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

**Note**
The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```json
{
   "repository": {
      "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
      "registryId": "111122223333",
      "repositoryName": "hello-world",
      "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
      "createdAt": "2023-03-09T10:39:01+00:00",
      "imageTagMutability": "MUTABLE",
      "imageScanningConfiguration": {
         "scanOnPush": true
      },
      "encryptionConfiguration": {
         "encryptionType": "AES256"
      }
   }
}
```

3. Copy the `repositoryUri` from the output in the previous step.

4. Run the `docker tag` command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - Replace `docker-image:test` with the name and `tag` of your Docker image.
- Replace `<ECRrepositoryUri>` with the `repositoryUri` that you copied. Make sure to include :latest at the end of the URI.

```bash
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Example:

```bash
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the `docker push` command to deploy your local image to the Amazon ECR repository. Make sure to include :latest at the end of the repository URI.

```bash
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include :latest at the end of the URI.

```bash
aws lambda create-function \\
  --function-name hello-world \\
  --package-type Image \\
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \\
  --role arn:aws:iam::111122223333:role/lambda-ex
```

**Note**

You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```bash
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:
9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the `update-function-code` command to deploy the image to the Lambda function.

**Using an alternative base image with the runtime interface client**

If you use an [OS-only base image](#) or an alternative base image, you must include the runtime interface client in your image. The runtime interface client extends the [Lambda runtime API](#), which manages the interaction between Lambda and your function code.

Install the runtime interface client for Java in your Dockerfile, or as a dependency in your project. For example, to install the runtime interface client using the Maven package manager, add the following to your `pom.xml` file:

```xml
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
    <version>2.3.2</version>
</dependency>
```

For package details, see [AWS Lambda Java Runtime Interface Client](#) in the Maven Central Repository. You can also review the runtime interface client source code in the [AWS Lambda Java Support Libraries](#) GitHub repository.

The following example demonstrates how to build a container image for Java using an Amazon Corretto image. Amazon Corretto is a no-cost, multiplatform, production-ready distribution of the Open Java Development Kit (OpenJDK). The Maven project includes the runtime interface client as a dependency.

**Prerequisites**

To complete the steps in this section, you must have the following:
Creating an image from an alternative base image

1. Create a Maven project. The following parameters are required:
   - **groupId** – The full package namespace of your application.
   - **artifactId** – Your project name. This becomes the name of the directory for your project.

   **Linux/macOS**

   ```
   mvn -B archetype:generate \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DgroupId=example \
    -DartifactId=myapp \
    -DinteractiveMode=false
   ```

   **PowerShell**

   ```
   mvn -B archetype:generate `\n    -DarchetypeArtifactId=maven-archetype-quickstart `\n    -DgroupId=example `\n    -DartifactId=myapp `\n    -DinteractiveMode=false
   ```

2. Open the project directory.

   ```
   cd myapp
   ```

3. Open the pom.xml file and replace the contents with the following. This file includes the `aws-lambda-java-runtime-interface-client` as a dependency. Alternatively, you can install the runtime interface client in the Dockerfile. However, the simplest approach is to include the library as a dependency.

Using a non-AWS base image
Using a non-AWS base image
4. Open the `myapp/src/main/java/com/example/myapp` directory, and find the `App.java` file. This is the code for the Lambda function. Replace the code with the following.

**Example function handler**

```java
package example;

public class App {
    public static String sayHello() {
        return "Hello world!";
    }
}
```

5. The `mvn -B archetype:generate` command from step 1 also generated a dummy test case in the `src/test` directory. For the purposes of this tutorial, skip over running tests by deleting this entire generated `/test` directory.

6. Navigate back to the project's root directory, and then create a new Dockerfile. The following example Dockerfile uses an Amazon Corretto image. Amazon Corretto is a no-cost, multiplatform, production-ready distribution of the OpenJDK.

   - Set the `FROM` property to the URI of the base image.
   - Set the `ENTRYPOINT` to the module that you want the Docker container to run when it starts. In this case, the module is the runtime interface client.
   - Set the `CMD` argument to the Lambda function handler.

**Example Dockerfile**

```Dockerfile
FROM public.ecr.aws/amazoncorretto/amazoncorretto:21 as base

# Configure the build environment
FROM base as build
RUN yum install -y maven
WORKDIR /src

# Cache and copy dependencies
ADD pom.xml .
RUN mvn dependency:go-offline dependency:copy-dependencies

# Compile the function
ADD ..
```
RUN mvn package

# Copy the function artifact and dependencies onto a clean base
FROM base
WORKDIR /function

COPY --from=build /src/target/dependency/*.jar ./
COPY --from=build /src/target/*.jar ./

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/bin/java", ":cp", ":/",
"com.amazonaws.services.lambda.runtime.api.client.AWSLambda"
]
# Pass the name of the function handler as an argument to the runtime
CMD [ "example.App::sayHello"
]

7. Build the Docker image with the `docker build` command. The following example names the image `docker-image` and gives it the test tag.

```bash
docker build --platform linux/amd64 -t docker-image:test .
```

**Note**

The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

**(Optional) Test the image locally**

Use the runtime interface emulator to locally test the image. You can build the emulator into your image or install it on your local machine.

**To install and run the runtime interface emulator on your local machine**

1. From your project directory, run the following command to download the runtime interface emulator (x86-64 architecture) from GitHub and install it on your local machine.
Linux/macOS

```
mkdir -p ~/.aws-lambda-rie && 
curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-
run-time-interface-emulator/releases/latest/download/aws-lambda-rie && 
chmod +x ~/.aws-lambda-rie/aws-lambda-rie
```

To install the arm64 emulator, replace the GitHub repository URL in the previous command with the following:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

To install the arm64 emulator, replace the $downloadLink with the following:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. Start the Docker image with the **docker run** command. Note the following:

- **docker-image** is the image name and **test** is the tag.
- `/usr/bin/java -cp './*' com.amazonaws.services.lambda.runtime.api.client.AWSLambda
  example.App::sayHello` is the ENTRYPOINT followed by the CMD from your Dockerfile.
Linux/macOS

docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p 9000:8080 \
  --entrypoint /aws-lambda/aws-lambda-rie \
  docker-image:test \
    /usr/bin/java -cp './*' \
  com.amazonaws.services.lambda.runtime.api.client.AWSLambda \
  example.App::sayHello

PowerShell

docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p 9000:8080 \
  --entrypoint /aws-lambda/aws-lambda-rie \
  docker-image:test \
    /usr/bin/java -cp './*' \
  com.amazonaws.services.lambda.runtime.api.client.AWSLambda \
  example.App::sayHello

This command runs the image as a container and creates a local endpoint at localhost:9000/2015-03-31/functions/function/invocations.

Note
If you built the Docker image for the ARM64 instruction set architecture, be sure to use the --platform linux/arm64 option instead of --platform linux/amd64.

3. Post an event to the local endpoint.

Linux/macOS

In Linux and macOS, run the following curl command:

curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{"payload":"hello world!"}"
```

**PowerShell**

In PowerShell, run the following `Invoke-WebRequest` command:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{}' -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType "application/json"
```

4. Get the container ID.

   `docker ps`

5. Use the `docker kill` command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

   `docker kill 3766c4ab331c`

**Deploying the image**

**To upload the image to Amazon ECR and create the Lambda function**

1. Run the `get-login-password` command to authenticate the Docker CLI to your Amazon ECR registry.
• Set the --region value to the AWS Region where you want to create the Amazon ECR repository.

• Replace 111122223333 with your AWS account ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a repository in Amazon ECR using the `create-repository` command.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

![Note](image)

The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```
{
   "repository": {
      "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
      "registryId": "111122223333",
      "repositoryName": "hello-world",
      "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
      "createdAt": "2023-03-09T10:39:01+00:00",
      "imageTagMutability": "MUTABLE",
      "imageScanningConfiguration": {
         "scanOnPush": true
      },
      "encryptionConfiguration": {
         "encryptionType": "AES256"
      }
   }
}
```

3. Copy the repositoryUri from the output in the previous step.

Using a non-AWS base image
4. Run the `docker tag` command to tag your local image into your Amazon ECR repository as the latest version. In this command:

- Replace `docker-image:test` with the name and `tag` of your Docker image.
- Replace `<ECRrepositoryUri>` with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

```bash
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Example:

```bash
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the `docker push` command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.

```bash
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. **Create an execution role** for the function, if you don’t already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```bash
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

**Note**

You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see [Amazon ECR cross-account permissions](#).

8. Invoke the function.
aws lambda invoke --function-name hello-world response.json

You should see a response like this:

```
{
   "ExecutedVersion": "$LATEST",
   "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the `update-function-code` command to deploy the image to the Lambda function.
Java Lambda function customization settings

This page describes settings specific to Java functions in AWS Lambda. You can use these settings to customize Java runtime startup behavior. This can reduce overall function latency and improve overall function performance, without having to modify any code.

Sections

• **JAVA_TOOL_OPTIONS environment variable**

JAVA_TOOL_OPTIONS environment variable

In Java, Lambda supports the JAVA_TOOL_OPTIONS environment variable to set additional command-line variables in Lambda. You can use this environment variable in various ways, such as to customize tiered-compilation settings. The next example demonstrates how to use the JAVA_TOOL_OPTIONS environment variable for this use case.

Example: Customize tiered compilation settings

Tiered compilation is a feature of the Java virtual machine (JVM). You can use specific tiered compilation settings to make best use of the JVM's just-in-time (JIT) compilers. Typically, the C1 compiler is optimized for fast start-up time. The C2 compiler is optimized for best overall performance, but it also uses more memory and takes a longer time to achieve it.

There are 5 different levels of tiered compilation. At Level 0, the JVM interprets Java byte code. At Level 4, the JVM uses the C2 compiler to analyze profiling data collected during application startup. Over time, it monitors code usage to identify the best optimizations.

Customizing the tiered compilation level can help you reduce Java function cold start latency. For example, set the tiered compilation level to 1 to have the JVM use the C1 compiler. This compiler quickly produces optimized native code but it doesn't generate any profiling data and never uses the C2 compiler.

In the Java 17 runtime, the JVM flag for tiered compilation is set to stop at level 1 by default. For the Java 11 runtime and below, you can set the tiered compilation level to 1 by doing the following steps:

To customize tiered compilation settings (console)

1. Open the Functions page in the Lambda console.
2. Choose a Java function that you want to customize tiered compilation for.

3. Choose the Configuration tab, then choose Environment variables in the left menu.

4. Choose Edit.

5. Choose Add environment variable.

6. For the key, enter JAVA_TOOL_OPTIONS. For the value, enter 
    -XX:+TieredCompilation -XX:TieredStopAtLevel=1.

7. Choose Save.

**Note**

You can also use Lambda SnapStart to mitigate cold start issues. SnapStart uses cached snapshots of your execution environment to significantly improve start-up performance. For more information about SnapStart features, limitations, and supported regions, see Improving startup performance with Lambda SnapStart.
Example: Customizing GC behavior using JAVA_TOOL_OPTIONS

Java 11 runtimes use the Serial garbage collector (GC) for garbage collection. By default, Java 17 runtimes also use the Serial GC. However, with Java 17 you can also use the JAVA_TOOL_OPTIONS environment variable to change the default GC. You can choose between the Parallel GC and Shenandoah GC.

For example, if your workload uses more memory and multiple CPUs, consider using the Parallel GC for better performance. You can do this by appending the following to the value of your JAVA_TOOL_OPTIONS environment variable:

-XX:+UseParallelGC
AWS Lambda context object in Java

When Lambda runs your function, it passes a context object to the handler. This object provides methods and properties that provide information about the invocation, function, and execution environment.

**Context methods**

- `getRemainingTimeInMillis()` – Returns the number of milliseconds left before the execution times out.
- `getFunctionName()` – Returns the name of the Lambda function.
- `getFunctionVersion()` – Returns the version of the function.
- `getInvokedFunctionArn()` – Returns the Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `getMemoryLimitInMB()` – Returns the amount of memory that's allocated for the function.
- `getAwsRequestId()` – Returns the identifier of the invocation request.
- `getLogGroupName()` – Returns the log group for the function.
- `getLogStreamName()` – Returns the log stream for the function instance.
- `getIdentity()` – (mobile apps) Returns information about the Amazon Cognito identity that authorized the request.
- `getClientContext()` – (mobile apps) Returns the client context that's provided to Lambda by the client application.
- `getLogger()` – Returns the logger object for the function.

The following example shows a function that uses the context object to access the Lambda logger.

**Example** *Handler.java*

```java
package example;
import com.amazonaws.services.lambda.runtime.Context
import com.amazonaws.services.lambda.runtime.RequestHandler
import com.amazonaws.services.lambda.runtime.LambdaLogger

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, String>{
```
Gson gson = new GsonBuilder().setPrettyPrinting().create();

@Override
public String handleRequest(Map<String, String> event, Context context) {
    LambdaLogger logger = context.getLogger();
    String response = new String("200 OK");
    // log execution details
    logger.log("ENVIRONMENT VARIABLES: " + gson.toJson(System.getenv()));
    logger.log("CONTEXT: " + gson.toJson(context));
    // process event
    logger.log("EVENT: " + gson.toJson(event));
    logger.log("EVENT TYPE: " + event.getClass().toString());
    return response;
}

The function serializes the context object into JSON and records it in its log stream.

Example log output

```
START RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Version: $LATEST
...
CONTEXT:
{
    "memoryLimit": 512,
    "awsRequestId": "6bc28136-xmpl-4365-b021-0ce6b2e64ab0",
    "functionName": "java-console",
    ...
}
...
END RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0
REPORT RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Duration: 198.50 ms Billed Duration: 200 ms Memory Size: 512 MB Max Memory Used: 90 MB Init Duration: 524.75 ms
```

The interface for the context object is available in the [aws-lambda-java-core](https://aws.amazon.com) library. You can implement this interface to create a context class for testing. The following example shows a context class that returns dummy values for most properties and a working test logger.

Example `src/test/java/example/TestContext.java`

```
package example;
```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.CognitoIdentity;
import com.amazonaws.services.lambda.runtime.ClientContext;
import com.amazonaws.services.lambda.runtime.LambdaLogger

public class TestContext implements Context{
    public TestContext() {} 
    public String getAwsRequestId(){
        return new String("495b12a8-xmpl-4eca-8168-160484189f99");
    }
    public String getLogGroupName(){
        return new String("/aws/lambda/my-function");
    }
    ...
    public LambdaLogger getLogger(){
        return new TestLogger();
    }
}

For more information on logging, see AWS Lambda function logging in Java.

Context in sample applications

The GitHub repository for this guide includes sample applications that demonstrate the use of the context object. Each sample application includes scripts for easy deployment and cleanup, an AWS Serverless Application Model (AWS SAM) template, and supporting resources.

Sample Lambda applications in Java

- **java17-examples** – A Java function that demonstrates how to use a Java record to represent an input event data object.
- **java-basic** – A collection of minimal Java functions with unit tests and variable logging configuration.
- **java-events** – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the aws-lambda-java-events library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- **s3-java** – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
• **Use API Gateway to invoke a Lambda function** – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.

All of the sample applications have a test context class for unit tests. The `java-basic` application shows you how to use the context object to get a logger. It uses SLF4J and Log4J 2 to provide a logger that works for local unit tests.
AWS Lambda automatically monitors Lambda functions and sends log entries to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation and other output from your function’s code to the log stream. For more information about CloudWatch Logs, see Using Amazon CloudWatch logs with AWS Lambda.

To output logs from your function code, you can use methods on java.lang.System, or any logging module that writes to stdout or stderr.

### Sections
- Creating a function that returns logs
- Using Lambda advanced logging controls with Java
- Advanced logging with Log4j2 and SLF4J
- Other tools and libraries
- Using Powertools for AWS Lambda (Java) and AWS SAM for structured logging
- Using the Lambda console
- Using the CloudWatch console
- Using the AWS Command Line Interface (AWS CLI)
- Deleting logs
- Sample logging code

### Creating a function that returns logs
To output logs from your function code, you can use methods on java.lang.System, or any logging module that writes to stdout or stderr. The aws-lambda-java-core library provides a logger class named LambdaLogger that you can access from the context object. The logger class supports multiline logs.

The following example uses the LambdaLogger logger provided by the context object.

**Example Handler.java**

```java
// Handler value: example.Handler
public class Handler implements RequestHandler<Object, String>{
```
Gson gson = new GsonBuilder().setPrettyPrinting().create();
@Override
public String handleRequest(Object event, Context context)
{
    LambdaLogger logger = context.getLogger();
    String response = new String("SUCCESS");
    // log execution details
    logger.log("ENVIRONMENT VARIABLES: "+ gson.toJson(System.getenv()));
    logger.log("CONTEXT: " + gson.toJson(context));
    // process event
    logger.log("EVENT: " + gson.toJson(event));
    return response;
}

Example log format

START RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Version: $LATEST
ENVIRONMENT VARIABLES:
{
    "_HANDLER": "example.Handler",
    "AWS_EXECUTION_ENV": "AWS_Lambda_java8",
    "AWS_LAMBDA_FUNCTION_MEMORY_SIZE": "512",
    ...
}
CONTEXT:
{
    "memoryLimit": 512,
    "awsRequestId": "6bc28136-xmpl-4365-b021-0ce6b2e64ab0",
    "functionName": "java-console",
    ...
}
EVENT:
{
    "records": [
        {
            "messageId": "19dd0b57-xmpl-4ac1-bd88-01bb068cb78",
            "receiptHandle": "MessageReceiptHandle",
            "body": "Hello from SQS!",
            ...
        }
    ]
}
The Java runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details:

REPORT line data fields

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TraceId** – For traced requests, the AWS X-Ray trace ID.
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

Using Lambda advanced logging controls with Java

To give you more control over how your functions' logs are captured, processed, and consumed, you can configure the following logging options for supported Java runtimes:

- **Log format** - select between plain text and structured JSON format for your function's logs
- **Log level** - for logs in JSON format, choose the detail level of the logs Lambda sends to CloudWatch, such as ERROR, DEBUG, or INFO
- **Log group** - choose the CloudWatch log group your function sends logs to

For more information about these logging options, and instructions on how to configure your function to use them, see the section called “Configuring advanced logging controls for your Lambda function”.

To use the log format and log level options with your Java Lambda functions, see the guidance in the following sections.
Using structured JSON log format with Java

If you select JSON for your function's log format, Lambda will send logs output using the LambdaLogger class as structured JSON. Each JSON log object contains at least four key value pairs with the following keys:

- "timestamp" - the time the log message was generated
- "level" - the log level assigned to the message
- "message" - the contents of the log message
- "AWSrequestId" - the unique request ID for the function invocation

Depending on the logging method you use, log outputs from your function captured in JSON format can also contain additional key value pairs.

To assign a level to logs you create using the LambdaLogger logger, you need to provide a LogLevel argument in your logging command as shown in the following example.

**Example Java logging code**

```java
LambdaLogger logger = context.getLogger();
logger.log("This is a debug log", LogLevel.DEBUG);
```

This log output by this example code would be captured in CloudWatch Logs as follows:

**Example JSON log record**

```json
{
    "timestamp":"2023-11-01T00:21:51.358Z",
    "level":"DEBUG",
    "message":"This is a debug log",
    "AWSrequestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

If you don't assign a level to your log output, Lambda will automatically assign it the level INFO.

If your code already uses another logging library to produce JSON structured logs, you don’t need to make any changes. Lambda doesn’t double-encode any logs that are already JSON encoded. Even if you configure your function to use the JSON log format, your logging outputs appear in CloudWatch in the JSON structure you define.
Using log-level filtering with Java

For AWS Lambda to filter your application logs according to their log level, your function must use JSON formatted logs. You can achieve this in two ways:

- Create log outputs using the standard LambdaLogger and configure your function to use JSON log formatting. Lambda then filters your log outputs using the “level” key value pair in the JSON object described in the section called “Using structured JSON log format with Java”. To learn how to configure your function's log format, see the section called “Configuring advanced logging controls for your Lambda function”.

- Use another logging library or method to create JSON structured logs in your code that include a “level” key value pair defining the level of the log output. You can use any logging library that can write JSON logs to stdout or stderr. For example, you can use Powertools for AWS Lambda or the Log4j2 package to generate JSON structured log outputs from your code. See the section called “Using Powertools for AWS Lambda (Java) and AWS SAM for structured logging” and the section called “Advanced logging with Log4j2 and SLF4J” to learn more.

When you configure your function to use log-level filtering, you must select from the following options for the level of logs you want Lambda to send to CloudWatch Logs:

<table>
<thead>
<tr>
<th>Log level</th>
<th>Standard usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACE (most detail)</td>
<td>The most fine-grained information used to trace the path of your code's execution</td>
</tr>
<tr>
<td>DEBUG</td>
<td>Detailed information for system debugging</td>
</tr>
<tr>
<td>INFO</td>
<td>Messages that record the normal operation of your function</td>
</tr>
<tr>
<td>WARN</td>
<td>Messages about potential errors that may lead to unexpected behavior if unaddressed</td>
</tr>
<tr>
<td>ERROR</td>
<td>Messages about problems that prevent the code from performing as expected</td>
</tr>
<tr>
<td>FATAL (least detail)</td>
<td>Messages about serious errors that cause the application to stop functioning</td>
</tr>
</tbody>
</table>
For Lambda to filter your function's logs, you must also include a "timestamp" key value pair in your JSON log output. The time must be specified in valid [RFC 3339](https://tools.ietf.org/html/rfc3339) timestamp format. If you don't supply a valid timestamp, Lambda will assign the log the level INFO and add a timestamp for you.

Lambda sends logs of the selected level and lower to CloudWatch. For example, if you configure a log level of WARN, Lambda will send logs corresponding to the WARN, ERROR, and FATAL levels.

### Advanced logging with Log4j2 and SLF4J

**Note**

AWS Lambda does not include Log4j2 in its managed runtimes or base container images. These are therefore not affected by the issues described in CVE-2021-44228, CVE-2021-45046, and CVE-2021-45105.

For cases where a customer function includes an impacted Log4j2 version, we have applied a change to the Lambda Java managed runtimes and base container images that helps to mitigate the issues in CVE-2021-44228, CVE-2021-45046, and CVE-2021-45105. As a result of this change, customers using Log4J2 may see an additional log entry, similar to "Transforming org/apache/logging/log4j/core/lookup/JndiLookup (java.net.URLClassLoader@...)". Any log strings that reference the jndi mapper in the Log4J2 output will be replaced with "Patched JndiLookup::lookup()".

Independent of this change, we strongly encourage all customers whose functions include Log4j2 to update to the latest version. Specifically, customers using the aws-lambda-java-log4j library in their functions should update to version 1.5.0 (or later), and redeploy their functions. This version updates the underlying Log4j2 utility dependencies to version 2.17.0 (or later). The updated aws-lambda-java-log4j binary is available at the [Maven repository](https://mvnrepository.com/artifact/software.amazon.awslambda/java-log4j) and its source code is available in [Github](https://github.com/software-amazon-aws/aws-lambda-java-log4j).

Lastly, take note that any libraries related to **aws-lambda-java-log4j (v1.0.0 or 1.0.1)** should **not** be used under any circumstance. These libraries are related to version 1.x of log4j which went end of life in 2015. The libraries are not supported, not maintained, not patched, and have known security vulnerabilities.

To customize log output, support logging during unit tests, and log AWS SDK calls, use Apache Log4j2 with SLF4J. Log4j is a logging library for Java programs that enables you to configure log levels and use appender libraries. SLF4J is a facade library that lets you change which library you use without changing your function code.
To add the request ID to your function’s logs, use the appender in the `aws-lambda-java-log4j2` library.

**Example** [src/main/resources/log4j2.xml – Appender configuration](#)

```xml
<Configuration>
  <Appenders>
    <Lambda name="Lambda" format="${env:AWS_LAMBDA_LOG_FORMAT:-TEXT}">
      <LambdaTextFormat>
        <PatternLayout>
          <pattern>%d{yyyy-MM-dd HH:mm:ss} %X{AWSRequestId} %-5p %c{1} - %m%n</pattern>
        </PatternLayout>
      </LambdaTextFormat>
    </Lambda>
  </Appenders>
  <Loggers>
    <Root level="${env:AWS_LAMBDA_LOG_LEVEL:-INFO}">
      <AppenderRef ref="Lambda"/>
    </Root>
    <Logger name="software.amazon.awssdk" level="WARN"/>
    <Logger name="software.amazon.awssdk.request" level="DEBUG"/>
  </Loggers>
</Configuration>
```

You can decide how your Log4j2 logs are configured for either plain text or JSON outputs by specifying a layout under the `<LambdaTextFormat>` and `<LambdaJSONFormat>` tags.

In this example, in text mode, each line is prepended with the date, time, request ID, log level, and class name. In JSON mode, the `<JsonTemplateLayout>` is used with a configuration that ships together with the `aws-lambda-java-log4j2` library.

SLF4J is a facade library for logging in Java code. In your function code, you use the SLF4J logger factory to retrieve a logger with methods for log levels like `info()` and `warn()`. In your build configuration, you include the logging library and SLF4J adapter in the classpath. By changing the libraries in the build configuration, you can change the logger type without changing your function code. SLF4J is required to capture logs from the SDK for Java.

In the following example code, the handler class uses SLF4J to retrieve a logger.
Example **src/main/java/example/HandlerS3.java** – Logging with SLF4J

```java
package example;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;

import static org.apache.logging.log4j.CloseableThreadContext.put;

public class HandlerS3 implements RequestHandler<S3Event, String>{
    private static final Logger logger = LoggerFactory.getLogger(HandlerS3.class);

    @Override
    public String handleRequest(S3Event event, Context context) {
        for(var record : event.getRecords()) {
            try (var loggingCtx = put("awsRegion", record.getAwsRegion())) {
                loggingCtx.put("eventName", record.getEventName());
                loggingCtx.put("bucket", record.getS3().getBucket().getName());
                loggingCtx.put("key", record.getS3().getObject().getKey());

                logger.info("Handling s3 event");
            }
            return "Ok";
        }
    }
}
```

This code produces log outputs like the following:

**Example log format**

```json
{
    "timestamp": "2023-11-15T16:56:00.815Z",
    "level": "INFO",
    "message": "Handling s3 event",
    "logger": "example.HandlerS3",
    "AWSSRequestId": "0bced576-3936-4e5a-9dcd-db9477b77f97",
}
```
The build configuration takes runtime dependencies on the Lambda appender and SLF4J adapter, and implementation dependencies on Log4j2.

**Example `build.gradle` – Logging dependencies**

```gradle
dependencies {
  ...  
  'com.amazonaws:aws-lambda-java-log4j2:[1.6.0,)',
  'com.amazonaws:aws-lambda-java-events:[3.11.3,)',
  'org.apache.logging.log4j:log4j-layout-template-json:[2.17.1,)',
  'org.apache.logging.log4j:log4j-slf4j2-impl:[2.19.0,)',
  ... 
}
```

When you run your code locally for tests, the context object with the Lambda logger is not available, and there's no request ID for the Lambda appender to use. For example test configurations, see the sample applications in the next section.

### Other tools and libraries

**Powertools for AWS Lambda (Java)** is a developer toolkit to implement Serverless best practices and increase developer velocity. The [Logging utility](#) provides a Lambda optimized logger which includes additional information about function context across all your functions with output structured as JSON. Use this utility to do the following:

- Capture key fields from the Lambda context, cold start and structures logging output as JSON
- Log Lambda invocation events when instructed (disabled by default)
- Print all the logs only for a percentage of invocations via log sampling (disabled by default)
- Append additional keys to structured log at any point in time
- Use a custom log formatter (Bring Your Own Formatter) to output logs in a structure compatible with your organization's Logging RFC
Using Powertools for AWS Lambda (Java) and AWS SAM for structured logging

Follow the steps below to download, build, and deploy a sample Hello World Java application with integrated Powertools for AWS Lambda (Java) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Java 11
- AWS CLI version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World Java template.

   ```bash
   sam init --app-template hello-world-powertools-java --name sam-app --package-type Zip --runtime java11 --no-tracing
   ```

2. Build the app.

   ```bash
   cd sam-app && sam build
   ```

3. Deploy the app.

   ```bash
   sam deploy --guided
   ```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.
Note

For HelloWorldFunction may not have authorization defined, Is this okay?, make sure to enter y.

5. Get the URL of the deployed application:

```bash
aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6.Invoke the API endpoint:

```bash
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```json
{"message":"hello world"}
```

7. To get the logs for the function, run `sam logs`. For more information, see Working with logs in the AWS Serverless Application Model Developer Guide.

```bash
sam logs --stack-name sam-app
```

The log output looks like this:

```
2023/02/03/[LATEST]851411a899b545eea2cffe4a4cfb0c8 2023-02-03T09:24:34.095000
INIT_START Runtime Version: java:11.v15   Runtime Version ARN: arn:aws:lambda:eu-central-1::runtime:0a253e3b7a1cc9ce404bc435eeb2ad358d8f64338a618d0c224fe509403583ca
2023/02/03/[LATEST]851411a899b545eea2cffe4a4cfb0c8 2023-02-03T09:24:34.114000
   Picked up JAVA_TOOL_OPTIONS: -XX:+TieredCompilation -XX:TieredStopAtLevel=1
2023/02/03/[LATEST]851411a899b545eea2cffe4a4cfb0c8 2023-02-03T09:24:34.793000
   Transforming org/apache/logging/log4j/core/lookup/JndiLookup
(lambdainternal.CustomerClassLoader@1a6c5a9e)
2023/02/03/[LATEST]851411a899b545eea2cffe4a4cfb0c8 2023-02-03T09:24:34.114000
   Picked up JAVA_TOOL_OPTIONS: -XX:+TieredCompilation -XX:TieredStopAtLevel=1
2023/02/03/[LATEST]851411a899b545eea2cffe4a4cfb0c8 2023-02-03T09:24:34.793000
   Transforming org/apache/logging/log4j/core/lookup/JndiLookup
(lambdainternal.CustomerClassLoader@1a6c5a9e)
2023/02/03/[LATEST]851411a899b545eea2cffe4a4cfb0c8 2023-02-03T09:24:35.252000
   START RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765 Version: $LATEST
2023/02/03/[LATEST]851411a899b545eea2cffe4a4cfb0c8 2023-02-03T09:24:36.531000 {
   "_aws": {
      "Timestamp": 1675416276051,
      "CloudWatchMetrics": []
   }
```
"Namespace": "sam-app-powerools-java",
"Metrics": [
{
"Name": "ColdStart",
"Unit": "Count"
}
],
"Dimensions": [
[
"Service",
"FunctionName"
]
]
},
"function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
"traceId": "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
"FunctionName": "sam-app-HelloWorldFunction-y9Iu1FLJJBGD",
"functionVersion": "$LATEST",
"ColdStart": 1.0,
"Service": "service_undefined",
"logStreamId": "2023/02/03/[$LATEST]851411a899b545eea2cffeea4cfbec81",
"executionEnvironment": "AWS_Lambda_java11"
}
8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```bash
sam delete
```
Managing log retention

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which CloudWatch automatically deletes the logs. To set up log retention, add the following to your AWS SAM template:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Using the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded **Code** editor, you will find logs in the **execution results**. When you use the console test feature to invoke a function, you'll find **Log output** in the **Details** section.

Using the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

**To view logs on the CloudWatch console**

1. Open the **Log groups page** on the CloudWatch console.
2. Choose the log group for your function (**/aws/lambda/your-function-name**).
3. Choose a log stream.

Each log stream corresponds to an **instance of your function**. A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple
concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see Error processor sample application for AWS Lambda.

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- AWS Command Line Interface (AWS CLI) version 2
- AWS CLI – Quick configuration with **aws configure**

You can use the **AWS CLI** to retrieve logs for an invocation using the **--log-type** command option. The response contains a **LogResult** field that contains up to 4 KB of base64-encoded logs from the invocation.

**Example retrieve a log ID**

The following example shows how to retrieve a log ID from the **LogResult** field for a function named **my-function**.

```bash
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```json
{
    "StatusCode": 200,
    "LogResult":
    "U1RBULQgUmVxdWVzdE1kOiA4N2QwNDRiOC1mMTU0LTEzTgtOGNkYS0yOTc0YzV1NGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

**Example decode the logs**

In the same command prompt, use the **base64** utility to decode the logs. The following example shows how to retrieve base64-encoded logs for **my-function**.

```bash
aws lambda invoke --function-name my-function out --log-type Tail
```
The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-aws-index.html) in the *AWS Command Line Interface User Guide for Version 2*.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", ",_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"
ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms  Billed
Duration: 80 ms  Memory Size: 128 MB  Max Memory Used: 73 MB
```

The `base64` utility is available on Linux, macOS, and [Ubuntu on Windows](https://docs.aws.amazon.com/cli/latest/userguide/cli-aws-index.html). macOS users may need to use `base64 -D`.

**Example get-logs.sh script**

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-aws-index.html) in the *AWS Command Line Interface User Guide for Version 2*. 

---

Using the AWS Command Line Interface (AWS CLI) 739
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-name stream1 --limit 5

Example macOS and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```bash
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```bash
./get-logs.sh
```

You should see the following output:

```json
{
  "StatusCode": 200,
  "ExecutedVersion": "$LATEST"
}
{
  "events": [
    {,
      "timestamp": 1559763003171,
      "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST
",
      "ingestionTime": 1559763003309
    },
    {
      "timestamp": 1559763003173,
      "message": "2019-06-05T19:30:03.173Z 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r\n  AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
  \r ...",
      "ingestionTime": 1559763018353
    },
    {
      "timestamp": 1559763003173,
      "message": "2019-06-05T19:30:03.173Z 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": "value\"
}
",
      "ingestionTime": 1559763018353
    }
  ]
}
```
Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which logs are deleted automatically.

Sample logging code

The GitHub repository for this guide includes sample applications that demonstrate the use of various logging configurations. Each sample application includes scripts for easy deployment and cleanup, an AWS SAM template, and supporting resources.

Sample Lambda applications in Java

- **java17-examples** – A Java function that demonstrates how to use a Java record to represent an input event data object.
- **java-basic** – A collection of minimal Java functions with unit tests and variable logging configuration.
- **java-events** – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the **aws-lambda-java-events** library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
• **s3-java** – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.

• **Use API Gateway to invoke a Lambda function** – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.

The **java-basic** sample application shows a minimal logging configuration that supports logging tests. The handler code uses the `LambdaLogger` logger provided by the context object. For tests, the application uses a custom `TestLogger` class that implements the `LambdaLogger` interface with a Log4j2 logger. It uses SLF4J as a facade for compatibility with the AWS SDK. Logging libraries are excluded from build output to keep the deployment package small.
AWS Lambda function errors in Java

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

This page describes how to view Lambda function invocation errors for the Java runtime using the Lambda console and the AWS CLI.

Sections

- Syntax
- How it works
- Creating a function that returns exceptions
- Using the Lambda console
- Using the AWS Command Line Interface (AWS CLI)
- Error handling in other AWS services
- Sample applications
- What's next?

Syntax

In the following example, the runtime fails to deserialize the event into an object. The input is a valid JSON type, but it doesn't match the type expected by the handler method.

Example Lambda runtime error

```json
{
  "errorMessage": "An error occurred during JSON parsing",
  "errorType": "java.lang.RuntimeException",
  "stackTrace": [],
  "cause": {
    "errorMessage": "com.fasterxml.jackson.databind.exc.InvalidFormatException: Can not construct instance of java.lang.Integer from String value '1000,10': not a valid Integer value
 at [Source: lambdainternal.util.NativeMemoryAsInputStream@35fc6dc4; line: 1, column: 1] (through reference chain: java.lang.Object[0])",
    "errorType": "java.io.UncheckedIOException",
    "stackTrace": [],
    "cause": {
```
"errorMessage": "Can not construct instance of java.lang.Integer from String value '1000,10': not a valid Integer value\n at [Source: lambdainternal.util.NativeMemoryAsInputStream@35fc6dc4; line: 1, column: 1] (through reference chain: java.lang.Object[0])",
"errorType": "com.fasterxml.jackson.databind.exc.InvalidFormatException",
"stackTrace": [
"com.fasterxml.jackson.databind.exc.InvalidFormatException.from(InvalidFormatException.java:55)
"com.fasterxml.jackson.databind.exc.InvalidFormatException.from(InvalidFormatException.java:55)
"com.fasterxml.jackson.databind.exc.InvalidFormatException.from(InvalidFormatException.java:55)
"com.fasterxml.jackson.databind.exc.InvalidFormatException.from(InvalidFormatException.java:55)
...]
]
}
]

How it works

When you invoke a Lambda function, Lambda receives the invocation request and validates the permissions in your execution role, verifies that the event document is a valid JSON document, and checks parameter values.

If the request passes validation, Lambda sends the request to a function instance. The Lambda runtime environment converts the event document into an object, and passes it to your function handler.

If Lambda encounters an error, it returns an exception type, message, and HTTP status code that indicates the cause of the error. The client or service that invoked the Lambda function can handle the error programmatically, or pass it along to an end user. The correct error handling behavior depends on the type of application, the audience, and the source of the error.

The following list describes the range of status codes you can receive from Lambda.

2xx

A 2xx series error with a X-Amz-Function-Error header in the response indicates a Lambda runtime or function error. A 2xx series status code indicates that Lambda accepted the request, but instead of an error code, Lambda indicates the error by including the X-Amz-Function-Error header in the response.
A 4xx series error indicates an error that the invoking client or service can fix by modifying the request, requesting permission, or by retrying the request. 4xx series errors other than 429 generally indicate an error with the request.

A 5xx series error indicates an issue with Lambda, or an issue with the function's configuration or resources. 5xx series errors can indicate a temporary condition that can be resolved without any action by the user. These issues can't be addressed by the invoking client or service, but a Lambda function's owner may be able to fix the issue.

For a complete list of invocation errors, see InvokeFunction errors.

Creating a function that returns exceptions

You can create a Lambda function that displays human-readable error messages to users.

Note
To test this code, you need to include InputLengthException.java in your project src folder.

Example src/main/java/example/HandlerDivide.java – Runtime exception

```java
import java.util.List;

// Handler value: example.HandlerDivide
public class HandlerDivide implements RequestHandler<List<Integer>, Integer>{
    Gson gson = new GsonBuilder().setPrettyPrinting().create();
    @Override
    public Integer handleRequest(List<Integer> event, Context context)
    {
        LambdaLogger logger = context.getLogger();
        // process event
        if ( event.size() != 2 )
        {
            throw new InputLengthException("Input must be a list that contains 2 numbers.");
        }
    }
```
int numerator = event.get(0);
int denominator = event.get(1);
logger.log("EVENT: " + gson.toJson(event));
logger.log("EVENT TYPE: " + event.getClass().toString());
return numerator/denominator;
}

When the function throws InputLengthException, the Java runtime serializes it into the following document.

Example error document (whitespace added)

```
{
  "errorMessage":"Input must be a list that contains 2 numbers.",
  "errorType":"java.lang.InputLengthException",
  "stackTrace": [ 
    "example.HandlerDivide.handleRequest(HandlerDivide.java:23)",
    "example.HandlerDivide.handleRequest(HandlerDivide.java:14)"
  ]
}
```

In this example, InputLengthException is a RuntimeException. The RequestHandler interface does not allow checked exceptions. The RequestStreamHandler interface supports throwing IOException errors.

The return statement in the previous example can also throw a runtime exception.

```
return numerator/denominator;
```

This code can return an arithmetic error.

```
{"errorMessage":"/ by zero","errorType":"java.lang.ArithmeticException","stackTrace": [
  "example.HandlerDivide.handleRequest(HandlerDivide.java:28)",
  "example.HandlerDivide.handleRequest(HandlerDivide.java:23)"
]}
```

**Using the Lambda console**

You can invoke your function on the Lambda console by configuring a test event and viewing the output. The output is captured in the function's execution logs and, when active tracing is enabled, in AWS X-Ray.
To invoke a function on the Lambda console

1. Open the Functions page of the Lambda console.
2. Choose the function to test, and choose Test.
3. Under Test event, select New event.
4. Select a Template.
5. For Name, enter a name for the test. In the text entry box, enter the JSON test event.
6. Choose Save changes.
7. Choose Test.

The Lambda console invokes your function synchronously and displays the result. To see the response, logs, and other information, expand the Details section.

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- AWS Command Line Interface (AWS CLI) version 2
- AWS CLI – Quick configuration with aws configure

When you invoke a Lambda function in the AWS CLI, the AWS CLI splits the response into two documents. The AWS CLI response is displayed in your command prompt. If an error has occurred, the response contains a FunctionError field. The invocation response or error returned by the function is written to an output file. For example, output.json or output.txt.

The following invoke command example demonstrates how to invoke a function and write the invocation response to an output.txt file.

```bash
aws lambda invoke   \
  --function-name my-function   \
  --cli-binary-format raw-in-base64-out  \
  --payload '{"key1": "value1", "key2": "value2", "key3": "value3"}' output.txt
```

The cli-binary-format option is required if you're using AWS CLI version 2. To make this the default setting, run aws configure set cli-binary-format raw-in-base64-out. For more
Lambda also records up to 256 KB of the error object in the function's logs. For more information, see AWS Lambda function logging in Java.

**Error handling in other AWS services**

When another AWS service invokes your function, the service chooses the invocation type and retry behavior. AWS services can invoke your function on a schedule, in response to a lifecycle event on a resource, or to serve a request from a user. Some services invoke functions asynchronously and let Lambda handle errors, while others retry or pass errors back to the user.

For example, API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502 error code. To customize the error response, you must catch errors in your code and format a response in the required format.
We recommend using AWS X-Ray to determine the source of an error and its cause. X-Ray allows you to find out which component encountered an error, and see details about the errors. The following example shows a function error that resulted in a 502 response from API Gateway.

For more information, see Instrumenting Java code in AWS Lambda.

**Sample applications**

The GitHub repository for this guide includes sample applications that demonstrate the use of the errors. Each sample application includes scripts for easy deployment and cleanup, an AWS Serverless Application Model (AWS SAM) template, and supporting resources.

**Sample Lambda applications in Java**

- **java17-examples** – A Java function that demonstrates how to use a Java record to represent an input event data object.

- **java-basic** – A collection of minimal Java functions with unit tests and variable logging configuration.

- **java-events** – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the aws-lambda-java-events library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.

- **s3-java** – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.

- **Use API Gateway to invoke a Lambda function** – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.
The java-basic function includes a handler (HandlerDivide) that returns a custom runtime exception. The HandlerStream handler implements the RequestStreamHandler and can throw an IOException checked exception.

What's next?

- Learn how to show logging events for your Lambda function on the [Logging](#) page.
Instrumenting Java code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of two SDK libraries:

- **AWS Distro for OpenTelemetry (ADOT)** – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- **AWS X-Ray SDK for Java** – An SDK for generating and sending trace data to X-Ray.
- **Powertools for AWS Lambda (Java)** – A developer toolkit to implement Serverless best practices and increase developer velocity.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

⚠️ **Important**

The X-Ray and Powertools for AWS Lambda SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see [Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs](#).

Sections

- Using Powertools for AWS Lambda (Java) and AWS SAM for tracing
- Using Powertools for AWS Lambda (Java) and the AWS CDK for tracing
- Using ADOT to instrument your Java functions
- Using the X-Ray SDK to instrument your Java functions
- Activating tracing with the Lambda console
- Activating tracing with the Lambda API
- Activating tracing with AWS CloudFormation

Tracing 751
Using Powertools for AWS Lambda (Java) and AWS SAM for tracing

Follow the steps below to download, build, and deploy a sample Hello World Java application with integrated Powertools for AWS Lambda (Java) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Java 11
- AWS CLI version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World Java template.

   ```
sam init --app-template hello-world-powertools-java --name sam-app --package-type Zip --runtime java11 --no-tracing
   ```

2. Build the app.

   ```
cd sam-app && sam build
   ```

3. Deploy the app.

   ```
sam deploy --guided
   ```
4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.

### Note

For **HelloWorldFunction may not have authorization defined, Is this okay?, make sure to enter y.**

5. Get the URL of the deployed application:

```bash
aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```bash
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```json
{"message":"hello world"}
```

7. To get the traces for the function, run `sam traces`.

```bash
sam traces
```

The trace output looks like this:

New XRay Service Graph

Start time: 2023-02-03 14:31:48+01:00
End time: 2023-02-03 14:31:48+01:00
Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-y9Iu1FLJJBGD - Edges: []
Summary_statistics:
- total requests: 1
- ok count(2XX): 1
- error count(4XX): 0
- fault count(5XX): 0
- total response time: 5.587
Reference Id: 1 - client - sam-app-HelloWorldFunction-y9Iu1FLJJBGD - Edges: [0]
Summary_statistics:
- total requests: 0
- ok count(2XX): 0
- error count(4XX): 0
- fault count(5XX): 0
- total response time: 0

XRay Event [revision 3] at (2023-02-03T14:31:48.500000) with id (1-63dd0cc4-3c869dec72a586875da39777) and duration (5.603s)
- 5.587s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD [HTTP: 200]
- 4.053s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD
  - 1.181s - Initialization
  - 4.037s - Invocation
    - 1.981s - ## handleRequest
      - 1.840s - ## getPageContents
    - 0.000s - Overhead

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

sam delete

**Using Powertools for AWS Lambda (Java) and the AWS CDK for tracing**

Follow the steps below to download, build, and deploy a sample Hello World Java application with integrated [Powertools for AWS Lambda (Java)](link) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

**Prerequisites**

To complete the steps in this section, you must have the following:

- Java 11
- [AWS CLI version 2](link)
- [AWS CDK version 2](link)
• **AWS SAM CLI version 1.75 or later.** If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

## Deploy a sample AWS CDK application

1. Create a project directory for your new application.

   ```bash
   mkdir hello-world
   cd hello-world
   ```

2. Initialize the app.

   ```bash
cdk init app --language java
   ```

3. Create a maven project with the following command:

   ```bash
   mkdir app
   cd app
   mvn archetype:generate -DgroupId=helloworld -DartifactId=Function -
   DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
   ```

4. Open `pom.xml` in the `hello-world\app\Function` directory and replace the existing code with the following code that includes dependencies and maven plugins for Powertools.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>helloworld</groupId>
   <artifactId>Function</artifactId>
   <packaging>jar</packaging>
   <version>1.0-SNAPSHOT</version>
   <name>Function</name>
   <url>http://maven.apache.org</url>
   <properties>
     <maven.compiler.source>11</maven.compiler.source>
     <maven.compiler.target>11</maven.compiler.target>
     <log4j.version>2.17.2</log4j.version>
   </properties>
   <dependencies>
     <dependency>
     </dependency>
   </dependencies>
</project>
```
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
  <groupId>software.amazon.lambda</groupId>
  <artifactId>powertools-tracing</artifactId>
  <version>1.3.0</version>
</dependency>
<dependency>
  <groupId>software.amazon.lambda</groupId>
  <artifactId>powertools-metrics</artifactId>
  <version>1.3.0</version>
</dependency>
<dependency>
  <groupId>software.amazon.lambda</groupId>
  <artifactId>powertools-logging</artifactId>
  <version>1.3.0</version>
</dependency>
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-lambda-java-core</artifactId>
  <version>1.2.2</version>
</dependency>
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-lambda-java-events</artifactId>
  <version>3.11.1</version>
</dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>aspectj-maven-plugin</artifactId>
      <version>1.14.0</version>
      <configuration>
        <source>${maven.compiler.source}</source>
        <target>${maven.compiler.target}</target>
        <complianceLevel>${maven.compiler.target}</complianceLevel>
        <aspectLibraries>
          <aspectLibrary>
            <groupId>software.amazon.lambda</groupId>
          </aspectLibrary>
        </aspectLibraries>
      </configuration>
    </plugin>
  </plugins>
</build>
Using Powertools for AWS Lambda (Java) and the AWS CDK for tracing
5. Create the `hello-world/app/src/main/resource` directory and create `log4j.xml` for the log configuration.

```bash
mkdir -p src/main/resource
cd src/main/resource
touch log4j.xml
```

6. Open `log4j.xml` and add the following code.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Appenders>
        <Console name="JsonAppender" target="SYSTEM_OUT">
            <JsonTemplateLayout
                eventTemplateUri="classpath:LambdaJsonLayout.json" />
        </Console>
    </Appenders>
    <Loggers>
        <Logger name="JsonLogger" level="INFO" additivity="false">
            <AppenderRef ref="JsonAppender"/>
        </Logger>
        <Root level="info">
            <AppenderRef ref="JsonAppender"/>
        </Root>
    </Loggers>
</Configuration>
```

7. Open `App.java` from the `hello-world/app/Function/src/main/java/helloworld` directory and replace the existing code with the following code. This is the code for the Lambda function.
package helloworld;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Collectors;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import software.amazon.lambda.powertools.logging.Logging;
import software.amazon.lambda.powertools.metrics.Metrics;
import software.amazon.lambda.powertools.tracing.CaptureMode;
import static software.amazon.lambda.powertools.tracing.CaptureMode.*;

/**
 * Handler for requests to Lambda function.
 */
public class App implements RequestHandler<APIGatewayProxyRequestEvent,
APIGatewayProxyResponseEvent> {
    Logger log = LogManager.getLogger(App.class);

    @Logging(logEvent = true)
    @Tracing(captureMode = DISABLED)
    @Metrics(captureColdStart = true)
    public APIGatewayProxyResponseEvent handleRequest(final
    APIGatewayProxyRequestEvent input, final Context context) {
        Map<String, String> headers = new HashMap<>();
        headers.put("Content-Type", "application/json");
        headers.put("X-Custom-Header", "application/json");

        APIGatewayProxyResponseEvent response = new APIGatewayProxyResponseEvent()
            .withHeaders(headers);
        try {
            try {
8. Open HelloWorldStack.java from the hello-world/src/main/java/com/myorg directory and replace the existing code with the following code. This code uses Lambda Constructor and the ApiGatewayv2 Constructor to create a REST API and a Lambda function.

```java
package com.myorg;

import software.amazon.awscdk.*;
import software.amazon.awscdk.services.apigatewayv2.alpha.*;
import software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegration;
import software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegrationProps;
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.Tracing;
import software.amazon.awscdk.services.logs.RetentionDays;
```
import software.amazon.awscdk.services.s3/assets.AssetOptions;
import software.constructs.Construct;
import java.util.Arrays;
import java.util.List;
import static java.util.Collections.singletonList;
import static software.amazon.awscdk.BundlingOutput.ARCHIVED;

public class HelloWorldStack extends Stack {
    public HelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloWorldStack(final Construct scope, final String id, final StackProps props) {
        super(scope, id, props);

        List<String> functionPackagingInstructions = Arrays.asList(
            "/bin/sh",
            "-c",
            "cd Function " +
            "&& mvn clean install " +
            "&& cp /asset-input/Function/target/function.jar /asset-output/"
        );

        BundlingOptions.Builder builderOptions = BundlingOptions.builder()
            .command(functionPackagingInstructions)
            .image(Runtime.JAVA_11.getBundlingImage())
            .volumes(singletonList(
                DockerVolume.builder()
                    // Mount local .m2 repo to avoid download all the dependencies again inside the container
                    .hostPath(System.getProperty("user.home") + ".m2")
                    .containerPath("/root/.m2")
                    .build()
            ))
            .user("root")
            .outputType(ARCHIVED);

        Function function = new Function(this, "Function", FunctionProps.builder()
            .runtime(Runtime.JAVA_11)
            .code(Code.fromAsset("app", AssetOptions.builder())
            .build()

Using Powertools for AWS Lambda (Java) and the AWS CDK for tracing
.bundling(builderOptions
    .command(functionPackagingInstructions)
    .build())
.build())
.handler("helloworld.App::handleRequest")
.memorySize(1024)
.tracing(Tracing.ACTIVE)
.timeout(Duration.seconds(10))
.logRetention(RetentionDays.ONE_WEEK)
.build();

HttpApi httpApi = new HttpApi(this, "sample-api", HttpApiProps.builder()
    .apiName("sample-api")
    .build());

httpApi.addRoutes(AddRoutesOptions.builder()
    .path("/")
    .methods(singletonList(HttpMethod.GET))
    .integration(new HttpLambdaIntegration("function", function,
        HttpLambdaIntegrationProps.builder()
            .payloadFormatVersion(PayloadFormatVersion.VERSION_2_0)
            .build()))
    .build());

new CfnOutput(this, "HttpApi", CfnOutputProps.builder()
    .description("Url for Http Api")
    .value(httpApi.getApiEndpoint())
    .build());
}
}

9. Open pom.xml from the hello-world directory and replace the existing code with the following code.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
    xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.myorg</groupId>
    <artifactId>hello-world</artifactId>
```
Using Powertools for AWS Lambda (Java) and the AWS CDK for tracing
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>apigatewayv2-alpha</artifactId>
    <version>${cdk.version}-alpha.0</version>
</dependency>
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>apigatewayv2-integrations-alpha</artifactId>
    <version>${cdk.version}-alpha.0</version>
</dependency>
</dependencies>
</project>

10. Make sure you’re in the hello-world directory and deploy your application.

cdk deploy

11. Get the URL of the deployed application:

aws cloudformation describe-stacks --stack-name HelloWorldStack --query 'Stacks[0].Outputs[?OutputKey==`HttpApi`].OutputValue' --output text

12. Invoke the API endpoint:

curl -X GET <URL_FROM_PREVIOUS_STEP>

If successful, you’ll see this response:

{"message":"hello world"}

13. To get the traces for the function, run sam traces.

sam traces

The trace output looks like this:
New XRay Service Graph
- Start time: 2023-02-03 14:59:50+00:00
- End time: 2023-02-03 14:59:50+00:00
- Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYt0c9j - Edges: [1]
  - Summary_statistics:
    - total requests: 1
    - ok count(2XX): 1
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 0.924
- Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYt0c9j - Edges: []
  - Summary_statistics:
    - total requests: 1
    - ok count(2XX): 1
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 0.016
- Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYt0c9j - Edges: [0]
  - Summary_statistics:
    - total requests: 0
    - ok count(2XX): 0
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
- 0.924s - sam-app-HelloWorldFunction-YBg8yfYt0c9j [HTTP: 200]
- 0.016s - sam-app-HelloWorldFunction-YBg8yfYt0c9j
  - 0.739s - Initialization
  - 0.016s - Invocation
  - 0.013s - ## lambda_handler
  - 0.000s - ## app.hello
  - 0.000s - Overhead

14. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

cdk destroy
Using ADOT to instrument your Java functions

ADOT provides fully managed Lambda layers that package everything you need to collect telemetry data using the OTel SDK. By consuming this layer, you can instrument your Lambda functions without having to modify any function code. You can also configure your layer to do custom initialization of OTel. For more information, see Custom configuration for the ADOT Collector on Lambda in the ADOT documentation.

For Java runtimes, you can choose between two layers to consume:

- **AWS managed Lambda layer for ADOT Java (Auto-instrumentation Agent)** – This layer automatically transforms your function code at startup to collect tracing data. For detailed instructions on how to consume this layer together with the ADOT Java agent, see AWS Distro for OpenTelemetry Lambda Support for Java (Auto-instrumentation Agent) in the ADOT documentation.

- **AWS managed Lambda layer for ADOT Java** – This layer also provides built-in instrumentation for Lambda functions, but it requires a few manual code changes to initialize the OTel SDK. For detailed instructions on how to consume this layer, see AWS Distro for OpenTelemetry Lambda Support for Java in the ADOT documentation.

Using the X-Ray SDK to instrument your Java functions

To record data about calls that your function makes to other resources and services in your application, you can add the X-Ray SDK for Java to your build configuration. The following example shows a Gradle build configuration that includes the libraries that activate automatic instrumentation of AWS SDK for Java 2.x clients.

Example **build.gradle** – Tracing dependencies

```groovy
dependencies {
    implementation platform('software.amazon.awssdk:bom:2.15.0')
    implementation platform('com.amazonaws:aws-xray-recorder-sdk-bom:2.11.0')
    ...
    implementation 'com.amazonaws:aws-xray-recorder-sdk-core'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk-instrumentor'
    ...
}
```
After you add the correct dependencies and make the necessary code changes, activate tracing in your function's configuration via the Lambda console or the API.

**Activating tracing with the Lambda console**

To toggle active tracing on your Lambda function with the console, follow these steps:

**To turn on active tracing**

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose Configuration and then choose Monitoring and operations tools.
4. Choose Edit.
5. Under X-Ray, toggle on Active tracing.
6. Choose Save.

**Activating tracing with the Lambda API**

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

The following example AWS CLI command enables active tracing on a function named my-function.

```
aws lambda update-function-configuration --function-name my-function \ --tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.
Activating tracing with AWS CloudFormation

To activate tracing on an AWS::Lambda::Function resource in an AWS CloudFormation template, use the TracingConfig property.

Example **function-inline.yml** – Tracing configuration

```yaml
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
...
```

For an AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function resource, use the Tracing property.

Example **template.yml** – Tracing configuration

```yaml
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
...
```

Interpreting an X-Ray trace

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's execution role. Otherwise, add the AWSXRayDaemonWriteAccess policy to the execution role.

After you've configured active tracing, you can observe specific requests through your application. The X-Ray service graph shows information about your application and all its components. The following example from the error processor sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.
X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

**Note**

You cannot configure the X-Ray sampling rate for your functions.

When using active tracing, Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes for the primary function from the error processor sample application.

The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace...
with these two segments. Both are named **my-function**, but one has an origin of AWS::Lambda and the other has origin AWS::Lambda::Function.

This example expands the function segment to show its three subsegments:

- **Initialization** – Represents time spent loading your function and running *initialization code*. This subsegment only appears for the first event that each instance of your function processes.

- **Invocation** – Represents the time spent running your handler code.

- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

**Note**

*Lambda SnapStart* functions also include a Restore subsegment. The Restore subsegment shows the time it takes for Lambda to restore a snapshot, load the runtime (JVM), and run any *afterRestore* runtime hooks. The process of restoring snapshots can include time spent on activities outside the MicroVM. This time is reported in the Restore subsegment. You aren’t charged for the time spent outside the microVM to restore a snapshot.

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see [AWS X-Ray SDK for Java](https://docs.aws.amazon.com/xray/latest/dg/java.html) in the *AWS X-Ray Developer Guide*.
Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see AWS X-Ray pricing.

Storing runtime dependencies in a layer (X-Ray SDK)

If you use the X-Ray SDK to instrument AWS SDK clients your function code, your deployment package can become quite large. To avoid uploading runtime dependencies every time you update your function code, package the X-Ray SDK in a Lambda layer.

The following example shows an AWS::Serverless::LayerVersion resource that stores the AWS SDK for Java and X-Ray SDK for Java.

Example template.yml – Dependencies layer

Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/blank-java.zip
      Tracing: Active
      Layers:
        - !Ref libs
  ...

libs:
  Type: AWS::Serverless::LayerVersion
  Properties:
    LayerName: blank-java-lib
    Description: Dependencies for the blank-java sample app.
    ContentUri: build/blank-java-lib.zip
    CompatibleRuntimes:
      - java21

With this configuration, you update the library layer only if you change your runtime dependencies. Since the function deployment package contains only your code, this can help reduce upload times.
Creating a layer for dependencies requires build configuration changes to generate the layer archive prior to deployment. For a working example, see the java-basic sample application on GitHub.

**X-Ray tracing in sample applications (X-Ray SDK)**

The GitHub repository for this guide includes sample applications that demonstrate the use of X-Ray tracing. Each sample application includes scripts for easy deployment and cleanup, an AWS SAM template, and supporting resources.

**Sample Lambda applications in Java**

- [java17-examples](#) – A Java function that demonstrates how to use a Java record to represent an input event data object.
- [java-basic](#) – A collection of minimal Java functions with unit tests and variable logging configuration.
- [java-events](#) – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the [aws-lambda-java-events](#) library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- [s3-java](#) – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- **Use API Gateway to invoke a Lambda function** – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.

All of the sample applications have active tracing enabled for Lambda functions. For example, the s3-java application shows automatic instrumentation of AWS SDK for Java 2.x clients, segment management for tests, custom subsegments, and the use of Lambda layers to store runtime dependencies.
Creating a deployment package using Eclipse

This section shows how to package your Java code into a deployment package using Eclipse IDE and Maven plugin for Eclipse.

става

The AWS SDK Eclipse Toolkit provides an Eclipse plugin for you to both create a deployment package and also upload it to create a Lambda function. If you can use Eclipse IDE as your development environment, this plugin enables you to author Java code, create and upload a deployment package, and create your Lambda function. For more information, see the AWS Toolkit for Eclipse Getting Started Guide. For an example of using the toolkit for authoring Lambda functions, see Using Lambda with the AWS toolkit for Eclipse.

Topics

• Prerequisites
• Create and build a project

Prerequisites

Install the Maven Plugin for Eclipse.

1. Start Eclipse. From the Help menu in Eclipse, choose Install New Software.
2. In the Install window, type http://download.eclipse.org/technology/m2e/releases in the Work with: box, and choose Add.
3. Follow the steps to complete the setup.

Create and build a project

In this step, you start Eclipse and create a Maven project. You will add the necessary dependencies, and build the project. The build will produce a .jar, which is your deployment package.

1. Create a new Maven project in Eclipse.
   a. From the File menu, choose New, and then choose Project.
b. In the New Project window, choose Maven Project.

c. In the New Maven Project window, choose Create a simple project, and leave other default selections.

d. In the New Maven Project, Configure project windows, type the following Artifact information:

   - Group Id: doc-examples
   - Artifact Id: lambda-java-example
   - Version: 0.0.1-SNAPSHOT
   - Packaging: jar
   - Name: lambda-java-example

2. Add the aws-lambda-java-core dependency to the pom.xml file.

   It provides definitions of the RequestHandler, RequestStreamHandler, and Context interfaces. This allows you to compile code that you can use with AWS Lambda.

   a. Open the context (right-click) menu for the pom.xml file, choose Maven, and then choose Add Dependency.

   b. In the Add Dependency windows, type the following values:

      - Group Id: com.amazonaws
      - Artifact Id: aws-lambda-java-core
      - Version: 1.2.2

   Note

   If you are following other tutorial topics in this guide, the specific tutorials might require you to add more dependencies. Make sure to add those dependencies as required.

3. Add Java class to the project.

   a. Open the context (right-click) menu for the src/main/java subdirectory in the project, choose New, and then choose Class.

   b. In the New Java Class window, type the following values:
- **Package**: example
- **Name**: Hello

**Note**

If you are following other tutorial topics in this guide, the specific tutorials might recommend different package name or class name.

c. Add your Java code. If you are following other tutorial topics in this guide, add the provided code.

4. Build the project.

Open the context (right-click) menu for the project in **Package Explorer**, choose **Run As**, and then choose **Maven Build**. In the **Edit Configuration** window, type **package** in the **Goals** box.

**Note**

The resulting .jar, `lambda-java-example-0.0.1-SNAPSHOT.jar`, is not the final standalone .jar that you can use as your deployment package. In the next step, you add the Apache maven-shade-plugin to create the standalone .jar. For more information, go to [Apache Maven Shade plugin](https://maven.apache.org/maven-v3/shade-plugin.html).

5. Add the maven-shade-plugin plugin and rebuild.

The maven-shade-plugin will take artifacts (jars) produced by the **package** goal (produces customer code .jar), and create a standalone .jar that contains the compiled customer code, and the resolved dependencies from the `pom.xml`.

a. Open the context (right-click) menu for the `pom.xml` file, choose **Maven**, and then choose **Add Plugin**.

b. In the **Add Plugin** window, type the following values:

- **Group Id**: org.apache.maven.plugins
- **Artifact Id**: maven-shade-plugin
- **Version**: 3.2.2

Create and build a project
c. Now build again.

   This time we will create the jar as before, and then use the maven-shade-plugin to pull in dependencies to make the standalone .jar.

   i. Open the context (right-click) menu for the project, choose Run As, and then choose Maven build ....

   ii. In the Edit Configuration windows, type package shade:shade in the Goals box.

   iii. Choose Run.

   You can find the resulting standalone .jar (that is, your deployment package), in the /target subdirectory.

   Open the context (right-click) menu for the /target subdirectory, choose Show In, choose System Explorer, and you will find the lambda-java-example-0.0.1-SNAPSHOT.jar.
Java sample applications for AWS Lambda

The GitHub repository for this guide provides sample applications that demonstrate the use of Java in AWS Lambda. Each sample application includes scripts for easy deployment and cleanup, an AWS CloudFormation template, and supporting resources.

Sample Lambda applications in Java

- **java17-examples** – A Java function that demonstrates how to use a Java record to represent an input event data object.
- **java-basic** – A collection of minimal Java functions with unit tests and variable logging configuration.
- **java-events** – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the [aws-lambda-java-events](https://aws-lambda-java-events) library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- **s3-java** – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- **Use API Gateway to invoke a Lambda function** – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.

Running popular Java frameworks on Lambda

- **spring-cloud-function-samples** – An example from Spring that shows how to use the Spring Cloud Function framework to create AWS Lambda functions.
- **Serverless Spring Boot Application Demo** – An example that shows how to set up a typical Spring Boot application in a managed Java runtime with and without SnapStart, or as a GraalVM native image with a custom runtime.
- **Serverless Micronaut Application Demo** – An example that shows how to use Micronaut in a managed Java runtime with and without SnapStart, or as a GraalVM native image with a custom runtime. Learn more in the [Micronaut/Lambda guides](https://aws.amazon.com/lambda/micronaut/).
- **Serverless Quarkus Application Demo** – An example that shows how to use Quarkus in a managed Java runtime with and without SnapStart, or as a GraalVM native image with a custom runtime. Learn more in the [Quarkus/Lambda guide](https://aws.amazon.com/lambda/quarkus/) and [Quarkus/SnapStart guide](https://aws.amazon.com/lambda/snapstart/).
If you're new to Lambda functions in Java, start with the java-basic examples. To get started with Lambda event sources, see the java-events examples. Both of these example sets show the use of Lambda's Java libraries, environment variables, the AWS SDK, and the AWS X-Ray SDK. Each example uses a Lambda layer to package its dependencies separately from the function code, which speeds up deployment times. These examples require minimal setup and you can deploy them from the command line in less than a minute.
Building Lambda functions with Go

Go is implemented differently than other managed runtimes. Because Go compiles to native code, Lambda treats Go as a custom runtime. We recommend that you use the provided.al2023 or provided.al2 runtime to deploy Go functions to Lambda.

Go runtime support

The Go 1.x managed runtime for Lambda is based on the Amazon Linux AMI (AL1). Lambda will continue to support the Go 1.x managed runtime until maintenance support for the Amazon Linux AMI ends on December 31, 2023. If you're using the Go 1.x runtime, you must migrate your functions to provided.al2023 or provided.al2. No code changes are required for this migration. The only required changes relate to how you build your deployment package and which runtime you use to create your function. For more information, see the deployment instructions for .zip packages and container images.

<table>
<thead>
<tr>
<th>Name</th>
<th>Identifier</th>
<th>Operating system</th>
<th>Deprecation date</th>
<th>Block function create</th>
<th>Block function update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go 1.x</td>
<td>go1.x</td>
<td>Amazon Linux</td>
<td>Jan 8, 2024</td>
<td>Feb 8, 2024</td>
<td>Mar 12, 2024</td>
</tr>
</tbody>
</table>

The provided.al2023 and provided.al2 runtimes offer several advantages over go1.x, including support for the arm64 architecture (AWS Graviton2 processors), smaller binaries, and slightly faster invoke times.

OS-only

<table>
<thead>
<tr>
<th>Name</th>
<th>Identifier</th>
<th>Operating system</th>
<th>Deprecation date</th>
<th>Block function create</th>
<th>Block function update</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS-only Runtime</td>
<td>provided.al2023</td>
<td>Amazon Linux 2023</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Identifier</td>
<td>Operating system</td>
<td>Deprecation date</td>
<td>Block function create</td>
<td>Block function update</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
<td>------------------</td>
<td>------------------</td>
<td>-----------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>OS-only</td>
<td>provided.</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Runtime</td>
<td>a12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS-only</td>
<td>provided</td>
<td>Amazon Linux</td>
<td>Jan 8, 2024</td>
<td>Feb 8, 2024</td>
<td>Mar 12, 2024</td>
</tr>
<tr>
<td>Runtime</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Tools and libraries**

Lambda provides the following tools and libraries for the Go runtime:

- **AWS SDK for Go**: the official AWS SDK for the Go programming language.
- **[github.com/aws/aws-lambda-go/lambda](https://github.com/aws/aws-lambda-go/lambda)**: The implementation of the Lambda programming model for Go. This package is used by AWS Lambda to invoke your handler.
- **[github.com/aws/aws-lambda-go/lambdacontext](https://github.com/aws/aws-lambda-go/lambdacontext)**: Helpers for accessing context information from the context object.
- **[github.com/aws/aws-lambda-go/events](https://github.com/aws/aws-lambda-go/events)**: This library provides type definitions for common event source integrations.
- **[github.com/aws/aws-lambda-go/cmd/build-lambda-zip](https://github.com/aws/aws-lambda-go/cmd/build-lambda-zip)**: This tool can be used to create a .zip file archive on Windows.

For more information, see **aws-lambda-go** on GitHub.

Lambda provides the following sample applications for the Go runtime:

**Sample Lambda applications in Go**

- **go-al2** – A hello world function that returns the public IP address. This app uses the provided.a12 custom runtime.
- **blank-go** – A Go function that shows the use of Lambda's Go libraries, logging, environment variables, and the AWS SDK. This app uses the go1.x runtime.

**Topics**
• **AWS Lambda function handler in Go**
• **AWS Lambda context object in Go**
• **Deploy Go Lambda functions with .zip file archives**
• **Deploy Go Lambda functions with container images**
• **AWS Lambda function logging in Go**
• **AWS Lambda function errors in Go**
• **Instrumenting Go code in AWS Lambda**
• **Using environment variables**
AWS Lambda function handler in Go

The Lambda function handler is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

A Lambda function written in Go is authored as a Go executable. In your Lambda function code, you need to include the github.com/aws/aws-lambda-go/lambda package, which implements the Lambda programming model for Go. In addition, you need to implement handler function code and a main() function.

Example Go Lambda function

```go
package main

import (
    "context"
    "fmt"
    "github.com/aws/aws-lambda-go/lambda"
)

type MyEvent struct {
    Name string `json:"name"`
}

func HandleRequest(ctx context.Context, event *MyEvent) (*string, error) {
    if event == nil {
        return nil, fmt.Errorf("received nil event")
    }
    message := fmt.Sprintf("Hello %s!", event.Name)
    return &message, nil
}

func main() {
    lambda.Start(HandleRequest)
}
```

Here is a sample input for this function:

```json
{
    "name": "Jane"
}
```
Note the following:

- **package main**: In Go, the package containing `func main()` must always be named `main`.

- **import**: Use this to include the libraries your Lambda function requires. In this instance, it includes:
  - `context`: [AWS Lambda context object in Go](#).
  - `fmt`: The Go [Formatting](#) object used to format the return value of your function.
  - `github.com/aws/aws-lambda-go/lambda`: As mentioned previously, implements the Lambda programming model for Go.

- **func HandleRequest(ctx context.Context, event *MyEvent) (*string, error)**: This is the signature of your Lambda handler. It's the entry point for your Lambda function and contains the logic that is executed when your function is invoked. In addition, the parameters included denote the following:
  - `ctx context.Context`: Provides runtime information for your Lambda function invocation. `ctx` is the variable you declare to leverage the information available via [AWS Lambda context object in Go](#).
  - `event *MyEvent`: This is a parameter named `event` that points to `MyEvent`. It represents the input to the Lambda function.
  - `*string, error`: The handler returns two values. The first is a pointer to a string which contains the result of the Lambda function. The second is an error type, which is `nil` if there's no error and contains standard `error` information if something goes wrong. For more information on custom error handling, see [AWS Lambda function errors in Go](#).
  - `return &message, nil`: Returns two values. The first is a pointer to a string message, which is a greeting constructed using the Name field from the input event. The second value, `nil`, indicates that the function didn't encounter any errors.

- **func main()**: The entry point that runs your Lambda function code. This is required.

  By adding `lambda.Start(HandleRequest) between func main(){} code brackets, your Lambda function will be executed. Per Go language standards, the opening bracket, `{`, must be placed directly at the end of the `main` function signature.
Naming

provided.al2 runtime

For Go functions that use the provided.al2 runtime in a .zip deployment package, the executable file that contains your function code must be named bootstrap. If you're deploying the function with a .zip file, the bootstrap file must be at the root of the .zip file. For Go functions that use the provided.al2 runtime in a container image, you can use any name for the executable file.

You can use any name for the handler. To reference the handler value in your code, you can use the _HANDLER environment variable.

go1.x runtime

For Go functions that use the go1.x runtime, the executable file and the handler can share any name. For example, if you set the value of the handler to Handler, Lambda will call the main() function in the Handler executable file.

To change the function handler name in the Lambda console, on the Runtime settings pane, choose Edit.

Lambda function handler using structured types

In the example above, the input type was a simple string. But you can also pass in structured events to your function handler:

```go
package main

import (
    "fmt"
    "github.com/aws/aws-lambda-go/lambda"
)

type MyEvent struct {
    Name string `json:"What is your name?"`
    Age  int    `json:"How old are you?"
}

type MyResponse struct {
    Message string `json:"Answer"
`
func HandleLambdaEvent(event *MyEvent) (*MyResponse, error) {
    if event == nil {
        return nil, fmt.Errorf("received nil event")
    }
    return &MyResponse{Message: fmt.Sprintf("%s is %d years old!", event.Name, event.Age)}, nil
}

func main() {
    lambda.Start(HandleLambdaEvent)
}

Here is a sample input for this function:

{  
    "What is your name?": "Jim",
    "How old are you?": 33
}

The response looks like this:

{  
    "Answer": "Jim is 33 years old!"
}

To be exported, field names in the event struct must be capitalized. For more information on handling events from AWS event sources, see [aws-lambda-go/events](https://aws-lambda-go/events).

**Valid handler signatures**

You have several options when building a Lambda function handler in Go, but you must adhere to the following rules:

- The handler must be a function.
- The handler may take between 0 and 2 arguments. If there are two arguments, the first argument must implement `context.Context`.
- The handler may return between 0 and 2 arguments. If there is a single return value, it must implement `error`. If there are two return values, the second value must implement `error`. For
more information on implementing error-handling information, see [AWS Lambda function errors in Go](https://docs.aws.amazon.com/lambda/latest/dg/errors-go.html).

The following lists valid handler signatures. TIn and T0ut represent types compatible with the encoding/json standard library. For more information, see [func Unmarshal](https://golang.org/pkg/encoding/json/) to learn how these types are deserialized.

- func ()
- func () error
- func (TIn) error
- func () (TOut, error)
- func (context.Context) error
- func (context.Context, TIn) error
- func (context.Context) (TOut, error)
- func (context.Context, TIn) (TOut, error)

### Using global state

You can declare and modify global variables that are independent of your Lambda function's handler code. In addition, your handler may declare an init function that is executed when your handler is loaded. This behaves the same in AWS Lambda as it does in standard Go programs. A single instance of your Lambda function will never handle multiple events simultaneously.

#### Example Go function with global variables

**Note**

This code uses the AWS SDK for Go V2. For more information, see [Getting Started with the AWS SDK for Go V2](https://github.com/aws/aws-sdk-go-v2).
package main

import (
  "context"
  "github.com/aws/aws-lambda-go/lambda"
  "github.com/aws/aws-sdk-go-v2/config"
  "github.com/aws/aws-sdk-go-v2/service/s3"
  "github.com/aws/aws-sdk-go-v2/service/s3/types"
  "log"
)

var invokeCount int
var myObjects []types.Object

func init() {
  // Load the SDK configuration
  cfg, err := config.LoadDefaultConfig(context.TODO())
  if err != nil {
    log.Fatalf("Unable to load SDK config: %v", err)
  }

  // Initialize an S3 client
  svc := s3.NewFromConfig(cfg)

  // Define the bucket name as a variable so we can take its address
  bucketName := "examplebucket"
  input := &s3.ListObjectsV2Input{
    Bucket: &bucketName,
  }

  // List objects in the bucket
  result, err := svc.ListObjectsV2(context.TODO(), input)
  if err != nil {
    log.Fatalf("Failed to list objects: %v", err)
  }
  myObjects = result.Contents
}

func LambdaHandler(ctx context.Context) (int, error) {
  invokeCount++
  for i, obj := range myObjects {
  }
}
func main() {
    lambda.Start(LambdaHandler)
}
AWS Lambda context object in Go

When Lambda runs your function, it passes a context object to the [handler](#). This object provides methods and properties with information about the invocation, function, and execution environment.

The Lambda context library provides the following global variables, methods, and properties.

**Global variables**

- **FunctionName** – The name of the Lambda function.
- **FunctionVersion** – The [version](#) of the function.
- **MemoryLimitInMB** – The amount of memory that's allocated for the function.
- **LogGroupName** – The log group for the function.
- **LogStreamName** – The log stream for the function instance.

**Context methods**

- **Deadline** – Returns the date that the execution times out, in Unix time milliseconds.

**Context properties**

- **InvokedFunctionArn** – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- **AwsRequestID** – The identifier of the invocation request.
- **Identity** – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
- **ClientContext** – (mobile apps) Client context that's provided to Lambda by the client application.

**Accessing invoke context information**

Lambda functions have access to metadata about their environment and the invocation request. This can be accessed at [Package context](#). Should your handler include `context.Context` as a parameter, Lambda will insert information about your function into the context's Value
property. Note that you need to import the \texttt{lambdacontext} library to access the contents of the context.\texttt{Context} object.

```go
package main

import (
    "context"
    "log"
    "github.com/aws/aws-lambda-go/lambda"
    "github.com/aws/aws-lambda-go/lambdacontext"
)

func CognitoHandler(ctx context.Context) {
    lc, _ := lambdacontext.FromContext(ctx)
    log.Print(lc.Identity.CognitoIdentityPoolID)
}

func main() {
    lambda.Start(CognitoHandler)
}
```

In the example above, \texttt{l\_c} is the variable used to consume the information that the context object captured and \texttt{log.Print(lc.Identity.CognitoIdentityPoolID)} prints that information, in this case, the CognitoIdentityPoolID.

The following example introduces how to use the context object to monitor how long your Lambda function takes to complete. This allows you to analyze performance expectations and adjust your function code accordingly, if needed.

```go
package main

import (
    "context"
    "log"
    "time"
    "github.com/aws/aws-lambda-go/lambda"
)

func LongRunningHandler(ctx context.Context) (string, error) {
    deadline, _ := ctx.Deadline()
    deadline = deadline.Add(-100 * time.Millisecond)
    ...
```
timeoutChannel := time.After(time.Until(deadline))

for {
    select {
        case <- timeoutChannel:
            return "Finished before timing out.", nil
        default:
            log.Print("hello!")
            time.Sleep(50 * time.Millisecond)
    }
}

func main() {
    lambda.Start(LongRunningHandler)
}
Deploy Go Lambda functions with .zip file archives

Your AWS Lambda function's code consists of scripts or compiled programs and their dependencies. You use a deployment package to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives.

This page describes how to create a .zip file as your deployment package for the Go runtime, and then use the .zip file to deploy your function code to AWS Lambda using the AWS Management Console, AWS Command Line Interface (AWS CLI), and AWS Serverless Application Model (AWS SAM).

Note that Lambda uses POSIX file permissions, so you may need to set permissions for the deployment package folder before you create the .zip file archive.

Sections

- Creating a .zip file on macOS and Linux
- Creating a .zip file on Windows
- Creating and updating Go Lambda functions using .zip files
- Creating a Go layer for your dependencies

Creating a .zip file on macOS and Linux

The following steps show how to compile your executable using the `go build` command and create a .zip file deployment package for Lambda. Before compiling your code, make sure you have installed the `lambda` package from GitHub. This module provides an implementation of the runtime interface, which manages the interaction between Lambda and your function code. To download this library, run the following command.

```
go get github.com/aws/aws-lambda-go/lambda
```

If your function uses the AWS SDK for Go, download the standard set of SDK modules, along with any AWS service API clients required by your application. To learn how to install the SDK for Go, see Getting Started with the AWS SDK for Go V2.
**Using the provided runtime family (recommended)**

Go is implemented differently than other managed runtimes. Because Go compiles to native code, Lambda treats Go as a custom runtime. We recommend that you use the provided.al2023 or provided.al2 runtime to deploy Go functions to Lambda.

**To create a .zip deployment package (macOS/Linux)**

1. In the project directory that contains your application's `main.go` file, compile your executable. Note the following:
   - The executable must be named `bootstrap`. For more information, see [Naming](#).
   - Set your target *instruction set architecture*. The provided.al2023 and provided.al2 runtimes support both arm64 and x86_64.
   - You can use the optional `lambda.norpc` tag to exclude the Remote Procedure Call (RPC) component of the `lambda` library. The RPC component is only required when using the Go 1.x runtime. Excluding the RPC reduces the size of the deployment package.

   For the arm64 architecture:
   ```
   GOOS=linux GOARCH=arm64 go build -tags lambda.norpc -o bootstrap main.go
   ```

   For the x86_64 architecture:
   ```
   GOOS=linux GOARCH=amd64 go build -tags lambda.norpc -o bootstrap main.go
   ```

2. (Optional) You may need to compile packages with `CGO_ENABLED=0` set on Linux:

   ```
   GOOS=linux GOARCH=arm64 CGO_ENABLED=0 go build -o bootstrap -tags lambda.norpc main.go
   ```

   This command creates a stable binary package for standard C library (`libc`) versions, which may be different on Lambda and other devices.

3. Create a deployment package by packaging the executable in a .zip file.

   ```
   zip myFunction.zip bootstrap
   ```
4. Create the function. Note the following:

- The binary must be named *bootstrap*, but the handler name can be anything. For more information, see [Naming](#).
- The *--architectures* option is only required if you're using arm64. The default value is *x86_64*.
- For *--role*, specify the Amazon Resource Name (ARN) of the [execution role](#).

```bash
aws lambda create-function --function-name myFunction \
  --runtime provided.al2023 --handler bootstrap \
  --architectures arm64 \
  --role arn:aws:iam::111122223333:role/lambda-ex \
  --zip-file fileb://myFunction.zip
```

### Using the Go 1.x runtime (legacy)

The Go 1.x managed runtime for Lambda is based on the Amazon Linux AMI (AL1). Lambda will continue to support the Go 1.x managed runtime until maintenance support for the Amazon Linux AMI ends on December 31, 2023. If you're using the Go 1.x runtime, you must migrate your functions to *provided.al2023* or *provided.al2*.

### To create a `.zip` deployment package (MacOS/Linux)

1. In the project directory containing your application's `main.go` file, compile your executable using `go build`.

   ```bash
   GOOS=linux GOARCH=amd64 go build main.go
   ```

   Setting `GOOS` to `linux` and `GOARCH` to `amd64` ensures that the compiled executable is compatible with the [Go runtime](#), even if you compile it in a non-Linux environment or on a machine with an ARM processor.
2. (Optional) If your main package consists of multiple files, use the following `go build` command to compile the package:

```bash
GOOS=linux GOARCH=amd64 go build main
```

3. (Optional) You may need to compile packages with `CGO_ENABLED=0` set on Linux:

```bash
GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build main.go
```

This command creates a stable binary package for standard C library (libc) versions, which may be different on Lambda and other devices.

4. Create a deployment package by packaging the executable in a .zip file.

```bash
zip myFunction.zip main
```

5. Create the function. For `--role`, specify the Amazon Resource Name (ARN) of the execution role.

```bash
aws lambda create-function --function-name myFunction \
  --runtime go1.x --handler main \
  --role arn:aws:iam::111122223333:role/lambda-ex \
  --zip-file fileb://myFunction.zip
```

### Creating a .zip file on Windows

The following steps show how to download the `build-lambda-zip` tool for Windows from GitHub, compile your executable, and create a .zip deployment package.

⚠️ **Note**

If you have not already done so, you must install `git` and then add the `git` executable to your Windows `%PATH%` environment variable.

Before compiling your code, make sure you have installed the `lambda` library from GitHub. To download this library, run the following command.

```bash
go get github.com/aws/aws-lambda-go/lambda
```
If your function uses the AWS SDK for Go, download the standard set of SDK modules, along with any AWS service API clients required by your application. To learn how to install the SDK for Go, see Getting Started with the AWS SDK for Go V2.

Using the provided runtime family (recommended)

Go is implemented differently than other managed runtimes. Because Go compiles to native code, Lambda treats Go as a custom runtime. We recommend that you use the provided.al2023 or provided.al2 runtime to deploy Go functions to Lambda.

To create a .zip deployment package (Windows)

1. Download the build-lambda-zip tool from GitHub.

   ```
   go install github.com/aws/aws-lambda-go/cmd/build-lambda-zip@latest
   ```

2. Use the tool from your GOPATH to create a .zip file. If you have a default installation of Go, the tool is typically in %USERPROFILE%\Go\bin. Otherwise, navigate to where you installed the Go runtime and do one of the following:

   cmd.exe

   In cmd.exe, run one of the following, depending on your target instruction set architecture. The provided.al2023 and provided.al2 runtimes support both arm64 and x86_64.

   You can use the optional lambda.norpc tag to exclude the Remote Procedure Call (RPC) component of the lambda library. The RPC component is only required when using the Go 1.x runtime. Excluding the RPC reduces the size of the deployment package.

   **Example — For the x86_64 architecture**

   ```
   set GOOS=linux
   set GOARCH=amd64
   set CGO_ENABLED=0
   go build -tags lambda.norpc -o bootstrap main.go
   %USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
   ```

   **Example — For the arm64 architecture**

   ```
   set GOOS=linux
   set GOARCH=arm64
   ```
PowerShell

In PowerShell, run one of the following, depending on your target instruction set architecture. The provided.a12023 and provided.a12 runtimes support both arm64 and x86_64.

You can use the optional lambda.norpc tag to exclude the Remote Procedure Call (RPC) component of the lambda library. The RPC component is only required when using the Go 1.x runtime. Excluding the RPC reduces the size of the deployment package.

For the x86_64 architecture:

```powershell
$env:GOOS = "linux"
$env:GOARCH = "amd64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
%USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

For the arm64 architecture:

```powershell
$env:GOOS = "linux"
$env:GOARCH = "arm64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
%USERPROFILE%\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

3. Create the function. Note the following:

- The binary must be named bootstrap, but the handler name can be anything. For more information, see [Naming](#).
- The --architectures option is only required if you're using arm64. The default value is x86_64.
- For --role, specify the Amazon Resource Name (ARN) of the execution role.
Using the Go 1.x runtime (legacy)

The Go 1.x managed runtime for Lambda is based on the Amazon Linux AMI (AL1). Lambda will continue to support the Go 1.x managed runtime until maintenance support for the Amazon Linux AMI ends on December 31, 2023. If you're using the Go 1.x runtime, you must migrate your functions to `provided.al2023` or `provided.al2`.

To create a .zip deployment package (Windows)

1. Download the `build-lambda-zip` tool from GitHub.

   ```bash
   go install github.com/aws/aws-lambda-go/cmd/build-lambda-zip@latest
   ```

2. Use the tool from your GOPATH to create a .zip file. If you have a default installation of Go, the tool is typically in `%USERPROFILE%\Go\bin`. Otherwise, navigate to where you installed the Go runtime and do one of the following:

   **cmd.exe**

   In cmd.exe, run the following:

   ```bash
   set GOOS=linux
   set GOARCH=amd64
   set CGO_ENABLED=0
   go build main.go
   %USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip main
   ```

   **PowerShell**

   In PowerShell, run the following:

   ```bash
   $env:GOOS = "linux"
   $env:GOARCH = "amd64"
   $env:CGO_ENABLED = "0"
   go build main.go
   ```
3. Create the function. For `--role`, specify the Amazon Resource Name (ARN) of the execution role.

```bash
aws lambda create-function --function-name myFunction \
--runtime go1.x --handler main \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

**Creating and updating Go Lambda functions using .zip files**

Once you have created your .zip deployment package, you can use it to create a new Lambda function or update an existing one. You can deploy your .zip package using the Lambda console, the AWS Command Line Interface, and the Lambda API. You can also create and update Lambda functions using AWS Serverless Application Model (AWS SAM) and AWS CloudFormation.

The maximum size for a .zip deployment package for Lambda is 250 MB (unzipped). Note that this limit applies to the combined size of all the files you upload, including any Lambda layers.

**Creating and updating functions with .zip files using the console**

To create a new function, you must first create the function in the console, then upload your .zip archive. To update an existing function, open the page for your function, then follow the same procedure to add your updated .zip file.

If your .zip file is less than 50MB, you can create or update a function by uploading the file directly from your local machine. For .zip files greater than 50MB, you must upload your package to an Amazon S3 bucket first. For instructions on how to upload a file to an Amazon S3 bucket using the AWS Management Console, see [Getting started with Amazon S3](https://docs.aws.amazon.com/AmazonS3/latest/userguide/). To upload files using the AWS CLI, see [Move objects](https://docs.aws.amazon.com/cli/latest/reference/s3/move-object.html) in the [AWS CLI User Guide](https://docs.aws.amazon.com/cli/latest/index.html).

**Note**

You cannot convert an existing container image function to use a .zip archive. You must create a new function.
To create a new function (console)

1. Open the Functions page of the Lambda console and choose Create Function.
2. Choose Author from scratch.
3. Under Basic information, do the following:
   a. For Function name, enter the name for your function.
   b. For Runtime, choose one of the following, depending on which type of deployment package you created:
      - **Provide your own bootstrap on Amazon Linux 2023** (recommended): Choose this option to create the function using the provided.al2023 runtime.
      - **Go 1.x** (legacy): This runtime is based on the Amazon Linux AMI (AL1). Lambda will continue to support the Go 1.x runtime until maintenance support for the Amazon Linux AMI ends on December 31, 2023. We recommend that you use the provided.al2023 or provided.al2 runtime instead.
4. (Optional) Under Permissions, expand Change default execution role. You can create a new Execution role or use an existing one.
5. Choose Create function. Lambda creates a basic 'Hello world' function using your chosen runtime.

To upload a .zip archive from your local machine (console)

1. In the Functions page of the Lambda console, choose the function you want to upload the .zip file for.
2. Select the Code tab.
3. In the Code source pane, choose Upload from.
5. To upload the .zip file, do the following:
   a. Select Upload, then select your .zip file in the file chooser.
   b. Choose Open.
   c. Choose Save.
To upload a .zip archive from an Amazon S3 bucket (console)

1. In the Functions page of the Lambda console, choose the function you want to upload a new .zip file for.
2. Select the Code tab.
3. In the Code source pane, choose Upload from.
4. Choose Amazon S3 location.
5. Paste the Amazon S3 link URL of your .zip file and choose Save.

Creating and updating functions with .zip files using the AWS CLI

You can use the AWS CLI to create a new function or to update an existing one using a .zip file. Use the create-function and update-function-code commands to deploy your .zip package. If your .zip file is smaller than 50MB, you can upload the .zip package from a file location on your local build machine. For larger files, you must upload your .zip package from an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the AWS CLI User Guide.

Note

If you upload your .zip file from an Amazon S3 bucket using the AWS CLI, the bucket must be located in the same AWS Region as your function.

To create a new function using a .zip file with the AWS CLI, you must specify the following:

- The name of your function (--function-name)
- Your function's runtime (--runtime)
- The Amazon Resource Name (ARN) of your function's execution role (--role)
- The name of the handler method in your function code (--handler)

You must also specify the location of your .zip file. If your .zip file is located in a folder on your local build machine, use the --zip-file option to specify the file path, as shown in the following example command.

```
aws lambda create-function --function-name myFunction \  
--runtime provided.al2023 --handler bootstrap 
```
To specify the location of .zip file in an Amazon S3 bucket, use the `--code` option as shown in the following example command. You only need to use the `S3ObjectVersion` parameter for versioned objects.

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=myBucketName,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

To update an existing function using the CLI, you specify the the name of your function using the `--function-name` parameter. You must also specify the location of the .zip file you want to use to update your function code. If your .zip file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the `--s3-bucket` and `--s3-key` options as shown in the following example command. You only need to use the `--s3-object-version` parameter for versioned objects.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket myBucketName --s3-key myFileName.zip --s3-object-version myObjectVersion
```

Creating and updating functions with .zip files using the Lambda API

To create and update functions using a .zip file archive, use the following API operations:

- CreateFunction
- UpdateFunctionCode

Creating and updating functions with .zip files using AWS SAM

The AWS Serverless Application Model (AWS SAM) is a toolkit that helps streamline the process of building and running serverless applications on AWS. You define the resources for your application in a YAML or JSON template and use the AWS SAM command line interface (AWS SAM CLI) to build,
package, and deploy your applications. When you build a Lambda function from an AWS SAM template, AWS SAM automatically creates a .zip deployment package or container image with your function code and any dependencies you specify. To learn more about using AWS SAM to build and deploy Lambda functions, see Getting started with AWS SAM in the AWS Serverless Application Model Developer Guide.

You can also use AWS SAM to create a Lambda function using an existing .zip file archive. To create a Lambda function using AWS SAM, you can save your .zip file in an Amazon S3 bucket or in a local folder on your build machine. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the AWS CLI User Guide.

In your AWS SAM template, the AWS::Serverless::Function resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

- **PackageType** - set to Zip
- **CodeUri** - set to the function code's Amazon S3 URI, path to local folder, or FunctionCode object
- **Runtime** - Set to your chosen runtime

With AWS SAM, if your .zip file is larger than 50MB, you don’t need to upload it to an Amazon S3 bucket first. AWS SAM can upload .zip packages up to the maximum allowed size of 250MB (unzipped) from a location on your local build machine.

To learn more about deploying functions using .zip file in AWS SAM, see AWS::Serverless::Function in the AWS SAM Developer Guide.

**Example: Using AWS SAM to build a Go function with provided.al2023**

1. Create an AWS SAM template with the following properties:
   - **BuildMethod**: Specifies the compiler for your application. Use go1.x.
   - **Runtime**: Use provided.al2023.
   - **CodeUri**: Enter the path to your code.
   - **Architectures**: Use [arm64] for the arm64 architecture. For the x86_64 instruction set architecture, use [amd64] or remove the Architectures property.

**Example template.yaml**

```yaml
AWSTemplateFormatVersion: '2010-09-09'
```
2. Use the `sam build` command to compile the executable.

   ```bash
   sam build
   ```

3. Use the `sam deploy` command to deploy the function to Lambda.

   ```bash
   sam deploy --guided
   ```

Creating and updating functions with .zip files using AWS CloudFormation

You can use AWS CloudFormation to create a Lambda function using a .zip file archive. To create a Lambda function from a .zip file, you must first upload your file to an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see [Move objects](https://docs.aws.amazon.com/cli/latest/manual/awscli reference-manualघरेलू बाज़ार उद्योग ग्राफ।) in the [AWS CLI User Guide](https://docs.aws.amazon.com/cli/latest/reference/)

In your AWS CloudFormation template, the `AWS::Lambda::Function` resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

- **PackageType** - Set to `Zip`
- **Code** - Enter the Amazon S3 bucket name and the .zip file name in the `S3Bucket` and `S3Key` fields
- **Runtime** - Set to your chosen runtime

Creating a Go layer for your dependencies

Note

Using layers with functions in a compiled language like Go may not provide the same amount of benefit as with an interpreted language like Python. Since Go is a compiled language, your functions still have to manually load any shared assemblies into memory during the init phase, which can increase cold start times. Instead, we recommend including any shared code at compile time to take advantage of any built-in compiler optimizations.

The instructions in this section show you how to include your dependencies in a layer.

Lambda automatically detects any libraries in the /opt/lib directory, and any binaries in the /opt/bin directory. To ensure that Lambda properly finds your layer content, create a layer with the following structure:

```plaintext
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

After you package your layer, see the section called “Creating and deleting layers” and the section called “Adding layers” to complete your layer setup.
Deploy Go Lambda functions with container images

There are three ways to build a container image for a Go Lambda function:

- **Using an AWS OS-only base image**

  Go is implemented differently than other managed runtimes. Because Go compiles to native code, Lambda treats Go as a custom runtime. We recommend that you use the provided.al2023 or provided.al2 OS-only base image to build Go images for Lambda. To make the image compatible with Lambda, you must include the aws-lambda-go/lambda package in the image.

- **Using a non-AWS base image**

  You can use an alternative base image from another container registry, such as Alpine Linux or Debian. You can also use a custom image created by your organization. To make the image compatible with Lambda, you must include the aws-lambda-go/lambda package in the image.

- **Using the AWS base image for Go 1.x (legacy)**

  Lambda will continue to support the Go 1.x base image until maintenance support for the Amazon Linux AMI ends on December 31, 2023. We recommend that you use the provided.al2023 or provided.al2 OS-only base image instead of the Go 1.x base image.

Tip

To reduce the time it takes for Lambda container functions to become active, see **Use multi-stage builds** in the Docker documentation. To build efficient container images, follow the **Best practices for writing Dockerfiles**.

This page explains how to build, test, and deploy container images for Lambda.

**AWS base images for Go**

AWS provides the following base image for Go:
Go runtime interface client

The aws-lambda-go/lambda package includes an implementation of the runtime interface. For examples of how to use aws-lambda-go/lambda in your image, see Using an AWS OS-only base image or Using a non-AWS base image.

Using an AWS OS-only base image

Go is implemented differently than other managed runtimes. Because Go compiles to native code, Lambda treats Go as a custom runtime. We recommend that you use the provided.al2023 OS-only base image or provided.al2 OS-only base image to build container images for Go functions.

<table>
<thead>
<tr>
<th>Tags</th>
<th>Runtime</th>
<th>Operating system</th>
<th>Dockerfile</th>
<th>Deprecation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Go 1.x</td>
<td>Amazon Linux</td>
<td>Dockerfile for Go 1.x on GitHub</td>
<td>Jan 8, 2024</td>
</tr>
</tbody>
</table>

Amazon Elastic Container Registry repository: gallery.ecr.aws/lambda/go
Prerequisites

To complete the steps in this section, you must have the following:

- Go
- Docker
- AWS Command Line Interface (AWS CLI) version 2

Creating an image from the provided.al2023 base image

To build and deploy a Go function with the provided.al2023 base image

1. Create a directory for the project, and then switch to that directory.

   ```
   mkdir hello
   cd hello
   ```

2. Initialize a new Go module.

   ```
   go mod init example.com/hello-world
   ```

3. Add the lambda library as a dependency of your new module.

   ```
   go get github.com/aws/aws-lambda-go/lambda
   ```

4. Create a file named main.go and then open it in a text editor. This is the code for the Lambda function. You can use the following sample code for testing, or replace it with your own.

   ```go
   package main

   import (
   "context"
   "github.com/aws/aws-lambda-go/events"
   "github.com/aws/aws-lambda-go/lambda"
   )

   func handler(ctx context.Context, event events.APIGatewayProxyRequest) (events.APIGatewayProxyResponse, error) {
     response := events.APIGatewayProxyResponse{
       StatusCode: 200,
       Body:     """Hello from Lambda!"",
   ```

Using an AWS OS-only base image
5. Use a text editor to create a Dockerfile in your project directory. The following example Dockerfile uses a multi-stage build. This allows you to use a different base image in each step. You can use one image, such as a Go base image, to compile your code and build the executable binary. You can then use a different image, such as provided.al2023, in the final FROM statement to define the image that you deploy to Lambda. The build process is separated from the final deployment image, so the final image only contains the files needed to run the application.

You can use the optional lambda.norpc tag to exclude the Remote Procedure Call (RPC) component of the lambda library. The RPC component is only required when using the Go 1.x runtime. Excluding the RPC reduces the size of the deployment package.

**Example — Multi-stage build Dockerfile**

```Dockerfile
FROM golang:1.20 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
# Build with optional lambda.norpc tag
COPY main.go .
RUN go build -tags lambda.norpc -o main main.go
# Copy artifacts to a clean image
FROM public.ecr.aws/lambda/provided:al2023
COPY --from=build /helloworld/main ./main
ENTRYPOINT [ "./main" ]
```

**Note**
Make sure that the version of Go that you specify in your Dockerfile (for example, golang:1.20) is the same version of Go that you used to create your application.
6. Build the Docker image with the `docker build` command. The following example names the image `docker-image` and gives it the test tag.

```bash
docker build --platform linux/amd64 -t docker-image:test .
```

**Note**

The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

Use the runtime interface emulator to locally test your image. The runtime interface emulator is included in the provided.a2023 base image.

**To run the runtime interface emulator on your local machine**

1. Start the Docker image with the `docker run` command. Note the following:

   - `docker-image` is the image name and `test` is the tag.
   - `./main` is the ENTRYPOINT from your Dockerfile.

   ```bash
docker run -d -v ~/.aws-lambda-rie:/aws-lambda -p 9000:8080 \
   --entrypoint /aws-lambda/aws-lambda-rie \
   docker-image:test \
   ./main
```

   This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

2. From a new terminal window, post an event to the following endpoint using a `curl` command:

   ```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```
This command invokes the function with an empty event and returns a response. Some functions might require a JSON payload. Example:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d 
'{"payload":"hello world!"}"
```

3. Get the container ID.

```
docker ps
```

4. Use the `docker kill` command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

**Deploying the image**

**To upload the image to Amazon ECR and create the Lambda function**

1. Run the `get-login-password` command to authenticate the Docker CLI to your Amazon ECR registry.
   
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace `111122223333` with your AWS account ID.

   ```
   aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
   ```

2. Create a repository in Amazon ECR using the `create-repository` command.

   ```
   aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
   ```

Using an AWS OS-only base image
Note

The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```json
{
   "repository": {
      "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
      "registryId": "111122223333",
      "repositoryName": "hello-world",
      "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
      "createdAt": "2023-03-09T10:39:01+00:00",
      "imageTagMutability": "MUTABLE",
      "imageScanningConfiguration": {
         "scanOnPush": true
      },
      "encryptionConfiguration": {
         "encryptionType": "AES256"
      }
   }
}
```

3. Copy the `repositoryUri` from the output in the previous step.

4. Run the `docker tag` command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - Replace `docker-image:test` with the name and `tag` of your Docker image.
   - Replace `<ECRRepositoryUri>` with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

   ```bash
docker tag docker-image:test <ECRRepositoryUri>:latest
   ```

   Example:
5. Run the `docker push` command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. **Create an execution role** for the function, if you don’t already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

**Note**

You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.
To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the `update-function-code` command to deploy the image to the Lambda function.

**Using a non-AWS base image**

You can build a container image for Go from a non-AWS base image. The example Dockerfile in the following steps uses an Alpine base image.

You must include the `aws-lambda-go/lambda` package with your Go handler. This package implements the programming model for Go, including the runtime interface.

**Prerequisites**

To complete the steps in this section, you must have the following:

- Go
- Docker
- AWS Command Line Interface (AWS CLI) version 2

**Creating an image from an alternative base image**

**To build and deploy a Go function with an Alpine base image**

1. Create a directory for the project, and then switch to that directory.
   
   ```
   mkdir hello
   cd hello
   ```

2. Initialize a new Go module.
   
   ```
   go mod init example.com/hello-world
   ```

3. Add the `lambda` library as a dependency of your new module.
   
   ```
   go get github.com/aws/aws-lambda-go/lambda
   ```

4. Create a file named `main.go` and then open it in a text editor. This is the code for the Lambda function. You can use the following sample code for testing, or replace it with your own.

   ```
   package main
   ```
import (  
"context"
"github.com/aws/aws-lambda-go/events"
"github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, event events.APIGatewayProxyRequest) (events.APIGatewayProxyResponse, error) {  
response := events.APIGatewayProxyResponse{      
    StatusCode: 200,  
    Body:        "Hello from Lambda!",  
}  
return response, nil
}

func main() {  
lambda.Start(handler)
}

5. Use a text editor to create a Dockerfile in your project directory. The following example Dockerfile uses an Alpine base image.

Example Dockerfile

```
FROM golang:1.20.2-alpine3.16 as build
WORKDIR /helloworld

# Copy dependencies list
COPY go.mod go.sum ./

# Build
COPY main.go .
RUN go build -o main main.go

# Copy artifacts to a clean image
FROM alpine:3.16
COPY --from=build /helloworld/main /main
ENTRYPOINT [ "/main" ]
```

Note

Make sure that the version of Go that you specify in your Dockerfile (for example, golang:1.20) is the same version of Go that you used to create your application.
6. Build the Docker image with the `docker build` command. The following example names the image `docker-image` and gives it the test tag.

```bash
docker build --platform linux/amd64 -t docker-image:test.
```

**Note**

The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

Use the runtime interface emulator to locally test the image. You can build the emulator into your image or install it on your local machine.

**To install and run the runtime interface emulator on your local machine**

1. From your project directory, run the following command to download the runtime interface emulator (x86-64 architecture) from GitHub and install it on your local machine.

   **Linux/macOS**

   ```bash
   mkdir -p ~/.aws-lambda-rie && 
   chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   
   To install the arm64 emulator, replace the GitHub repository URL in the previous command with the following:
   
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```
PowerShell

```powershell
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

To install the arm64 emulator, replace the $downloadLink with the following:

```powershell
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
```

2. Start the Docker image with the `docker run` command. Note the following:

   - `docker-image` is the image name and `test` is the tag.
   - `/main` is the ENTRYPOINT from your Dockerfile.

Linux/macOS

```bash
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
    /main
```

PowerShell

```powershell
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p 9000:8080 ` \
    --entrypoint /aws-lambda/aws-lambda-rie ` \
    docker-image:test ` \
    /main
```
This command runs the image as a container and creates a local endpoint at localhost:9000/2015-03-31/functions/function/invocations.

**Note**

If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

3. Post an event to the local endpoint.

**Linux/macOS**

In Linux and macOS, run the following `curl` command:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```bash
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{"payload":"hello world!"}'
```

**PowerShell**

In PowerShell, run the following `Invoke-WebRequest` command:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{}" -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```powershell
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType "application/json"
```
4. Get the container ID.

```bash
docker ps
```

5. Use the `docker kill` command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```bash
docker kill 3766c4ab331c
```

**Deploying the image**

**To upload the image to Amazon ECR and create the Lambda function**

1. Run the `get-login-password` command to authenticate the Docker CLI to your Amazon ECR registry.

   • Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   
   • Replace `111122223333` with your AWS account ID.

   ```bash
   aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
   ```

2. Create a repository in Amazon ECR using the `create-repository` command.

   ```bash
   aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
   ```

**Note**

The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```json
{
   "repository": {
```
3. Copy the repositoryUri from the output in the previous step.

4. Run the `docker tag` command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - Replace `docker-image:test` with the name and tag of your Docker image.
   - Replace `<ECRrepositoryUri>` with the repositoryUri that you copied. Make sure to include :latest at the end of the URI.

   ```bash
docker tag docker-image:test <ECRrepositoryUri>:latest
```

   Example:

   ```bash
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the `docker push` command to deploy your local image to the Amazon ECR repository. Make sure to include :latest at the end of the repository URI.

   ```bash
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. **Create an execution role** for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.
7. Create the Lambda function. For ImageUri, specify the repository URI from earlier. Make sure to include :latest at the end of the URI.

```
aws lambda create-function
  --function-name hello-world
  --package-type Image
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
  --role arn:aws:iam::111122223333:role/lambda-ex
```

**Note**

You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see [Amazon ECR cross-account permissions](#).

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the `update-function-code` command to deploy the image to the Lambda function.

**Using the AWS base image for Go 1.x (legacy)**

We recommend that you use the provided.a12023 or provided.a12 [OS-only base image](#) instead of the [Go 1.x base image](#). Lambda will continue to support the Go 1.x base image until maintenance support for the Amazon Linux AMI ends on December 31, 2023. For more information about Amazon Linux AMI support, see [Amazon Linux AMI FAQs](#).
If you're using the Go 1.x base image, you must migrate your functions to provided.al2023 or provided.al2. No code changes are required for this migration. The only required changes relate to how you build your deployment package and which runtime you use to create your function. For more information, see Using an AWS OS-only base image.

**Prerequisites**

To complete the steps in this section, you must have the following:

- Go
- Docker
- AWS Command Line Interface (AWS CLI) version 2

**Creating an image from a base image**

1. Create a directory for the project, and then switch to that directory.

   ```bash
   mkdir hello
   cd hello
   ```

2. Initialize a new Go module.

   ```bash
   go mod init example.com/hello-world
   ```

3. Add the `lambda` library as a dependency of your new module.

   ```bash
   go get github.com/aws/aws-lambda-go/lambda
   ```

4. Create a file named `main.go` and then open it in a text editor. This is the code for the Lambda function. You can use the following sample code for testing, or replace it with your own.

   ```go
   package main

   import (   
     "context"   
     "github.com/aws/aws-lambda-go/events"   
     "github.com/aws/aws-lambda-go/lambda"
   )

   func handler(ctx context.Context, event events.APIGatewayProxyRequest) (events.APIGatewayProxyResponse, error) {
   ```
5. Build the Go project. This command creates an executable called `main`.

```
GOOS=linux GOARCH=amd64 go build -o main main.go
```

Setting `GOOS` to `linux` and `GOARCH` to `amd64` ensures that the compiled executable is compatible with the architecture of the Go base image, even if you compile it in a non-Linux environment.

6. Create a new Dockerfile. The example Dockerfile uses the following configuration:

- **FROM**: The URI of the base image that you want to use.
- **COPY**: Copies the function code into the `/var/task` directory in your image.
- **CMD**: The Lambda function handler or executable.

```
FROM public.ecr.aws/lambda/go:1

# Copy function handler executable
COPY main ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside of the Dockerfile)
CMD [ "main" ]
```

7. Build the Docker image with the `docker build` command. The following example names the image `docker-image` and gives it the test `tag`.

```
docker build --platform linux/amd64 -t docker-image:test .
```
### Note

The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

### (Optional) Test the image locally

1. Start the Docker image with the `docker run` command. In this example, `docker-image` is the image name and `test` is the tag.

   ```bash
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

   **Note**

   If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

2. From a new terminal window, post an event to the local endpoint.

   **Linux/macOS**

   In Linux and macOS, run the following `curl` command:

   ```bash
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:
PowerShell

In PowerShell, run the following Invoke-WebRequest command:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{}' -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType "application/json"
```

3. Get the container ID.

```
docker ps
```

4. Use the `docker kill` command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the `get-login-password` command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace `111122223333` with your AWS account ID.

```
```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com

2. Create a repository in Amazon ECR using the `create-repository` command.

```bash
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

**Note**
The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```json
{
   "repository": {
      "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
      "registryId": "111122223333",
      "repositoryName": "hello-world",
      "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
      "createdAt": "2023-03-09T10:39:01+00:00",
      "imageTagMutability": "MUTABLE",
      "imageScanningConfiguration": {
         "scanOnPush": true
      },
      "encryptionConfiguration": {
         "encryptionType": "AES256"
      }
   }
}
```

3. Copy the `repositoryUri` from the output in the previous step.

4. Run the `docker tag` command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - Replace `docker-image:test` with the name and tag of your Docker image.
• Replace `<ECRrepositoryUri>` with the repositoryUri that you copied. Make sure to include :latest at the end of the URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Example:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the `docker push` command to deploy your local image to the Amazon ECR repository. Make sure to include :latest at the end of the repository URI.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For ImageUri, specify the repository URI from earlier. Make sure to include :latest at the end of the URI.

```
aws lambda create-function
  --function-name hello-world
  --package-type Image
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
  --role arn:aws:iam::111122223333:role/lambda-ex
```

**Note**

You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see [Amazon ECR cross-account permissions](#).

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:
9. To see the output of the function, check the response.json file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the update-function-code command to deploy the image to the Lambda function.
AWS Lambda

Developer Guide

AWS Lambda function logging in Go
AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon
CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream
for each instance of your function. The Lambda runtime environment sends details about each
invocation to the log stream, and relays logs and other output from your function's code. For more
information, see Using Amazon CloudWatch logs with AWS Lambda.
This page describes how to produce log output from your Lambda function's code, or access logs
using the AWS Command Line Interface, the Lambda console, or the CloudWatch console.
Sections
• Creating a function that returns logs
• Using the Lambda console
• Using the CloudWatch console
• Using the AWS Command Line Interface (AWS CLI)
• Deleting logs

Creating a function that returns logs
To output logs from your function code, you can use methods on the fmt package, or any logging
library that writes to stdout or stderr. The following example uses the log package.
Example main.go – Logging
func handleRequest(ctx context.Context, event events.SQSEvent) (string, error) {
// event
eventJson, _ := json.MarshalIndent(event, "", " ")
log.Printf("EVENT: %s", eventJson)
// environment variables
log.Printf("REGION: %s", os.Getenv("AWS_REGION"))
log.Println("ALL ENV VARS:")
for _, element := range os.Environ() {
log.Println(element)
}

Example log format
START RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Version: $LATEST
Logging

829


The Go runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details.

**REPORT line data fields**

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TraceId** – For traced requests, the [AWS X-Ray trace ID](https://docs.aws.amazon.com/lambda/latest/dg/monitoring-xray.html).
• **SegmentId** – For traced requests, the X-Ray segment ID.
• **Sampled** – For traced requests, the sampling result.

**Using the Lambda console**

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded **Code** editor, you will find logs in the **execution results**. When you use the console test feature to invoke a function, you'll find **Log output** in the **Details** section.

**Using the CloudWatch console**

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

**To view logs on the CloudWatch console**

1. Open the **Log groups page** on the CloudWatch console.
2. Choose the log group for your function (**/aws/lambda/your-function-name**).
3. Choose a log stream.

Each log stream corresponds to an **instance of your function**. A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see **Error processor sample application for AWS Lambda**.

**Using the AWS Command Line Interface (AWS CLI)**

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

• **AWS Command Line Interface (AWS CLI) version 2**
• **AWS CLI – Quick configuration with aws configure**
You can use the AWS CLI to retrieve logs for an invocation using the --log-type command option. The response contains a LogResult field that contains up to 4 KB of base64-encoded logs from the invocation.

**Example retrieve a log ID**

The following example shows how to retrieve a log ID from the LogResult field for a function named `my-function`.

```bash
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{
   "StatusCode": 200,
   "LogResult":
      "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTEzTgtOGNkY20yOTc0YzV1NGZiMjEgVmVyc21vblw",
   "ExecutedVersion": "$LATEST"
}
```

**Example decode the logs**

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```bash
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --decode
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-commands.html) in the AWS Command Line Interface User Guide for Version 2.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-f5792818b6fe83688e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib,
```
The base64 utility is available on Linux, macOS, and Ubuntu on Windows. macOS users may need to use base64 -D.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses sed to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the get-log-events command.

Copy the contents of the following code sample and save in your Lambda project directory as get-logs.sh.

The cli-binary-format option is required if you're using AWS CLI version 2. To make this the default setting, run aws configure set cli-binary-format raw-in-base64-out. For more information, see AWS CLI supported global command line options in the AWS Command Line Interface User Guide for Version 2.

```bash
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --payload '{"key": "value"}' out
sed -i'' -e 's//\"/\"/g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-name stream1 --limit 5
```

Example macOS and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```bash
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.
You should see the following output:

```json
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf\tINFO\tENVIRONMENT VARIABLES\r{\r  "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
...\r \r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf\tINFO\tEVENT\r{\r  "key": "value"}
",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75 MB\t",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
}
```
Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which logs are deleted automatically.
AWS Lambda function errors in Go

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

This page describes how to view Lambda function invocation errors for the Go runtime using the Lambda console and the AWS CLI.

Sections

• Creating a function that returns exceptions
• How it works
• Using the Lambda console
• Using the AWS Command Line Interface (AWS CLI)
• Error handling in other AWS services
• What's next?

Creating a function that returns exceptions

The following code sample demonstrates custom error handling that raises an exception directly from a Lambda function and handles it directly. Note that custom errors in Go must import the errors module.

```go
package main

import (  
    "errors"
    "github.com/aws/aws-lambda-go/lambda"
)

func OnlyErrors() error {  
    return errors.New("something went wrong!")
}

func main() {  
    lambda.Start(OnlyErrors)
}
```

Which returns the following:
How it works

When you invoke a Lambda function, Lambda receives the invocation request and validates the permissions in your execution role, verifies that the event document is a valid JSON document, and checks parameter values.

If the request passes validation, Lambda sends the request to a function instance. The Lambda runtime environment converts the event document into an object, and passes it to your function handler.

If Lambda encounters an error, it returns an exception type, message, and HTTP status code that indicates the cause of the error. The client or service that invoked the Lambda function can handle the error programatically, or pass it along to an end user. The correct error handling behavior depends on the type of application, the audience, and the source of the error.

The following list describes the range of status codes you can receive from Lambda.

2xx

A 2xx series error with a X-Amz-Function-Error header in the response indicates a Lambda runtime or function error. A 2xx series status code indicates that Lambda accepted the request, but instead of an error code, Lambda indicates the error by including the X-Amz-Function-Error header in the response.

4xx

A 4xx series error indicates an error that the invoking client or service can fix by modifying the request, requesting permission, or by retrying the request. 4xx series errors other than 429 generally indicate an error with the request.

5xx

A 5xx series error indicates an issue with Lambda, or an issue with the function's configuration or resources. 5xx series errors can indicate a temporary condition that can be resolved without any action by the user. These issues can't be addressed by the invoking client or service, but a Lambda function's owner may be able to fix the issue.
For a complete list of invocation errors, see `InvokeFunction errors`.

**Using the Lambda console**

You can invoke your function on the Lambda console by configuring a test event and viewing the output. The output is captured in the function's execution logs and, when `active tracing` is enabled, in AWS X-Ray.

**To invoke a function on the Lambda console**

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to test, and choose **Test**.
3. Under **Test event**, select **New event**.
4. Select a **Template**.
5. For **Name**, enter a name for the test. In the text entry box, enter the JSON test event.
6. Choose **Save changes**.
7. Choose **Test**.

The Lambda console invokes your function **synchronously** and displays the result. To see the response, logs, and other information, expand the **Details** section.

**Using the AWS Command Line Interface (AWS CLI)**

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface (AWS CLI) version 2](#)
- [AWS CLI – Quick configuration with `aws configure`](#)

When you invoke a Lambda function in the AWS CLI, the AWS CLI splits the response into two documents. The AWS CLI response is displayed in your command prompt. If an error has occurred, the response contains a `FunctionError` field. The invocation response or error returned by the function is written to an output file. For example, `output.json` or `output.txt`.

The following `invoke` command example demonstrates how to invoke a function and write the invocation response to an `output.txt` file.
aws lambda invoke   \\
   --function-name my-function   \\
   --cli-binary-format raw-in-base64-out   \\
   --payload '{"key1": "value1", "key2": "value2", "key3": "value3"}' output.txt

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the AWS Command Line Interface User Guide for Version 2.

You should see the AWS CLI response in your command prompt:

```
{
   "StatusCode": 200,
   "FunctionError": "Unhandled",
   "ExecutedVersion": "$LATEST"
}
```

You should see the function invocation response in the `output.txt` file. In the same command prompt, you can also view the output in your command prompt using:

```
cat output.txt
```

You should see the invocation response in your command prompt.

**Error handling in other AWS services**

When another AWS service invokes your function, the service chooses the invocation type and retry behavior. AWS services can invoke your function on a schedule, in response to a lifecycle event on a resource, or to serve a request from a user. Some services invoke functions asynchronously and let Lambda handle errors, while others retry or pass errors back to the user.

For example, API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502 error code. To customize the error response, you must catch errors in your code and format a response in the required format.
We recommend using AWS X-Ray to determine the source of an error and its cause. X-Ray allows you to find out which component encountered an error, and see details about the errors. The following example shows a function error that resulted in a 502 response from API Gateway.

For more information, see [Instrumenting Go code in AWS Lambda](#).

**What's next?**

- Learn how to show logging events for your Lambda function on the [section called “Logging”](#) page.
Instrumenting Go code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of two SDK libraries:

- **AWS Distro for OpenTelemetry (ADOT)** – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- **AWS X-Ray SDK for Go** – An SDK for generating and sending trace data to X-Ray.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

⚠️ **Important**

The X-Ray and Powertools for AWS Lambda SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see [Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs](#).

Sections

- **Using ADOT to instrument your Go functions**
- **Using the X-Ray SDK to instrument your Go functions**
- **Activating tracing with the Lambda console**
- **Activating tracing with the Lambda API**
- **Activating tracing with AWS CloudFormation**
- **Interpreting an X-Ray trace**
Using ADOT to instrument your Go functions

ADOT provides fully managed Lambda layers that package everything you need to collect telemetry data using the OTel SDK. By consuming this layer, you can instrument your Lambda functions without having to modify any function code. You can also configure your layer to do custom initialization of OTel. For more information, see Custom configuration for the ADOT Collector on Lambda in the ADOT documentation.

For Go runtimes, you can add the AWS managed Lambda layer for ADOT Go to automatically instrument your functions. For detailed instructions on how to add this layer, see AWS Distro for OpenTelemetry Lambda Support for Go in the ADOT documentation.

Using the X-Ray SDK to instrument your Go functions

To record details about calls that your Lambda function makes to other resources in your application, you can also use the AWS X-Ray SDK for Go. To get the SDK, download the SDK from its GitHub repository with go get:

```go
go get github.com/aws/aws-xray-sdk-go
```

To instrument AWS SDK clients, pass the client to the `xray.AWS()` method. You can then trace calls by using the `WithContext` version of the method.

```go
svc := s3.New(session.New())
xray.AWS(svc.Client)
...
svc.ListBucketsWithContext(ctx aws.Context, input *ListBucketsInput)
```

After you add the correct dependencies and make the necessary code changes, activate tracing in your function's configuration via the Lambda console or the API.

Activating tracing with the Lambda console

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose Configuration and then choose Monitoring and operations tools.
4. Choose Edit.
5. Under X-Ray, toggle on Active tracing.
6. Choose Save.

Activating tracing with the Lambda API

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

The following example AWS CLI command enables active tracing on a function named my-function.

```bash
aws lambda update-function-configuration --function-name my-function --tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Activating tracing with AWS CloudFormation

To activate tracing on an AWS::Lambda::Function resource in an AWS CloudFormation template, use the TracingConfig property.

Example `function-inline.yml` – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
```
For an AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function resource, use the Tracing property.

**Example template.yml – Tracing configuration**

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
```

**Interpreting an X-Ray trace**

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's execution role. Otherwise, add the AWSXRayDaemonWriteAccess policy to the execution role.

After you've configured active tracing, you can observe specific requests through your application. The X-Ray service graph shows information about your application and all its components. The following example from the error processor sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.
X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

**Note**
You cannot configure the X-Ray sampling rate for your functions.

When using active tracing, Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes for the primary function from the error processor sample application.

The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of `AWS::Lambda` and the other has origin `AWS::Lambda::Function`.

---

**Interpreting an X-Ray trace**

---
This example expands the function segment to show its three subsegments:

- **Initialization** – Represents time spent loading your function and running initialization code. This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see the [AWS X-Ray SDK for Go](https://docs.aws.amazon.com/xray/latest/current/getting-started-gotcha-getting-started.html) in the *AWS X-Ray Developer Guide*.

### Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see [AWS X-Ray pricing](https://aws.amazon.com/xray/pricing/).
Using environment variables

To access environment variables in Go, use the `Getenv` function.

The following explains how to do this. Note that the function imports the `fmt` package to format the printed results and the `os` package, a platform-independent system interface that allows you to access environment variables.

```go
package main

import (
    "fmt"
    "os"
    "github.com/aws/aws-lambda-go/lambda"
)

func main()
    fmt.Printf("%s is %s. years old\n", os.Getenv("NAME"), os.Getenv("AGE"))
}
```

For a list of environment variables that are set by the Lambda runtime, see Defined runtime environment variables.
Building Lambda functions with C#

The following sections explain how common programming patterns and core concepts apply when authoring Lambda function code in C#.

You can run your .NET application in Lambda using the managed .NET 6 runtime, a custom runtime, or a container image. After your application code is compiled, you can deploy it to Lambda either as a .zip file or a container image.

Lambda provides the following runtimes for .NET languages:

### .NET

<table>
<thead>
<tr>
<th>Name</th>
<th>Identifier</th>
<th>Operating system</th>
<th>Deprecation date</th>
<th>Block function create</th>
<th>Block function update</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NET 7 (container only)</td>
<td>dotnet7</td>
<td>Amazon Linux 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NET 6</td>
<td>dotnet6</td>
<td>Amazon Linux 2</td>
<td>Nov 12, 2024</td>
<td>Jan 11, 2024</td>
<td>Feb 11, 2024</td>
</tr>
</tbody>
</table>

⚠️ **Note**

For end of support information about .NET Core 3.1, see the section called “Runtime deprecation policy”.

Topics

- Setting up your .NET development environment
- Lambda function handler in C#
- Build and deploy C# Lambda functions with .zip file archives
- Deploy .NET Lambda functions with container images
- .NET functions with native AOT compilation
- AWS Lambda context object in C#
- Lambda function logging in C#
- AWS Lambda function errors in C#
- Instrumenting C# code in AWS Lambda
- AWS Lambda function testing in C#
Setting up your .NET development environment

To develop and build your Lambda functions, you can use any of the commonly available .NET integrated development environments (IDEs), including Microsoft Visual Studio, Visual Studio Code, and JetBrains Rider. To simplify your development experience, AWS provides a set of .NET project templates, as well as the Amazon.Lambda.Tools command line interface (CLI).

Run the following .NET CLI commands to install these project templates and command line tools.

To install the project templates (.NET 6):

```bash
dotnet new --install Amazon.Lambda.Templates
```

To install the project templates (.NET 7):

```bash
dotnet new install Amazon.Lambda.Templates
```

To install the command line tools:

```bash
dotnet tool install -g Amazon.Lambda.Tools
```
Lambda function handler in C#

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

When your function is invoked and Lambda runs your function's handler method, it passes two arguments to your function. The first argument is the *event* object. When another AWS service invokes your function, the *event* object contains data about the event that caused your function to be invoked. For example, an *event* object from API Gateway contains information about the path, HTTP method, and HTTP headers. The exact event structure varies according to the AWS service invoking your function. See *Integrating other services* for more information about event formats for individual services.

Lambda also passes a *context* object to your function. This object contains information about the invocation, function, and execution environment. For more information, see the section called “Context”.

The native format for all Lambda events is streams of bytes representing the JSON formatted event. Unless your function input and output parameters are of type `System.IO.Stream`, you must serialize them. Specify the serializer you want to use by setting the `LambdaSerializer` assembly attribute. For more information, see the section called “Serialization in Lambda functions”.

Topics

- [.NET execution models for Lambda](#)
- Class library handlers
- Executable assembly handlers
- Serialization in Lambda functions
- Simplify function code with the Lambda Annotations framework
- Lambda function handler restrictions

.NET execution models for Lambda

There are two different execution models for running Lambda functions in .NET: the class library approach and the executable assembly approach.
In the **class library** approach, you provide Lambda with a string indicating the AssemblyName, ClassName, and Method of the function to be invoked. For more information about the format of this string, see the section called “Class library handlers”. During the function's initialization phase, your function's class is initialized, and any code in the constructor is run.

In the **executable assembly** approach, you use C# 9's top-level statements feature. This approach generates an executable assembly which Lambda runs whenever it receives an invoke command for your function. You provide Lambda only with the name of the executable assembly to run.

The following sections give example function code for these two approaches.

### Class library handlers

The following Lambda function code shows an example of a handler method (FunctionHandler) for a Lambda function which uses the class library approach. In this example function, Lambda receives an event from API Gateway that invokes the function. The function reads a record from a database and returns the record as part of the API Gateway response.

```csharp
[assembly:
]
namespace GetProductHandler;
public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
```
When you create a Lambda function, you need to provide Lambda with information about your function's handler in the form of a **handler string**. This tells Lambda which method in your code to run when your function is invoked. In C#, the format of the handler string when using the class library approach is as follows:

**ASSEMBLY::TYPE::METHOD**, where:

- **ASSEMBLY** is the name of the .NET assembly file for your application. If you use the Amazon.Lambda.Tools CLI to build your application and you don't set the assembly name using the AssemblyName property in the .csproj file, then ASSEMBLY is simply the name of your .csproj file.
- **TYPE** is the full name of the handler type, which consists of the Namespace and the ClassName.
- **METHOD** is the name of the function handler method in your code.

For the example code shown, if the assembly is named GetProductHandler, then the handler string would be `GetProductHandler::GetProductHandler.Function::FunctionHandler`.

**Executable assembly handlers**

In the following example, the Lambda function is defined as an executable assembly. The handler method in this code is named `Handler`. When using executable assemblies, the Lambda runtime must be bootstrapped. To do this, you use the `LambdaBootstrapBuilder.Create` method. This method takes as inputs the method your function uses as the handler and the Lambda serializer to use.

For more information about using top-level statements, see [Introducing the .NET 6 runtime for AWS Lambda](https://aws.amazon.com/blogs/compute/introducing-the-net-6-runtime-for-aws-lambda) on the AWS compute blog.

```csharp
namespace GetProductHandler;

IDatabaseRepository repo = new DatabaseRepository();

await LambdaBootstrapBuilder.Create<APIGatewayProxyRequest>(Handler, new DefaultLambdaJsonSerializer());
```
async Task<APIGatewayProxyResponse> Handler(APIGatewayProxyRequest apigProxyEvent, ILambdaContext context)
{
    var id = input.PathParameters["id"];

    var databaseRecord = await this.repo.GetById(id);

    return new APIGatewayProxyResponse
    {
        StatusCode = (int)HttpStatusCode.OK,
        Body = JsonSerializer.Serialize(databaseRecord)
    };
}

When using executable assemblies, the handler string that tells Lambda how to run your code is the name of the assembly. In this example, that would be GetProductHandler.

**Serialization in Lambda functions**

If your Lambda function uses input or output types other than a Stream object, you must add a serialization library to your application. You can implement serialization either using the standard reflection based serialization provided by System.Text.Json and Newtonsoft.Json, or by using **source generated serialization**.

**Using source generated serialization**

Source generated serialization is a feature of .NET 6 that allows serialization code to be generated at compile time. It removes the need for reflection and can improve the performance of your function. To use source generated serialization in your function, do the following:

- Create a new partial class that inherits from JsonSerializerContext, adding JsonSerializerable attributes for all types that require serialization or deserialization.
- Configure the LambdaSerializer to use a SourceGeneratorLambdaJsonSerializer<T>.
- Update any manual serialization or deserialization in your application code to use the newly created class.

An example function using source generated serialization is shown in the following code.
[assembly:
LambdaSerializer(typeof(SourceGeneratorLambdaJsonSerializer<CustomSerializer>))]

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord,
                                              CustomSerializer.Default.Product)
        };
    }
}

[JsonSerializable(typeof(APIGatewayProxyRequest))]
[JsonSerializable(typeof(APIGatewayProxyResponse))]
[JsonSerializable(typeof(Product))]
public partial class CustomSerializer : JsonSerializerContext
{
}

Note

If you want to use native ahead of time compilation (AOT) with Lambda, you must use source generated serialization.
Using reflection-based serialization


You can also create your own serialization library by implementing the ILambdaSerializer interface, which is available as part of the Amazon.Lambda.Core library. This interface defines two methods:

- T Deserialize<T>(Stream requestStream);
  
  You implement this method to deserialize the request payload from the Invoke API into the object that is passed to your Lambda function handler.

- T Serialize<T>(T response, Stream responseStream);
  
  You implement this method to serialize the result returned from your Lambda function handler into the response payload that the Invoke API operation returns.

Simplify function code with the Lambda Annotations framework

Lambda Annotations is a framework for .NET 6 which simplifies writing Lambda functions using C#. With the Annotations framework, you can replace much of the code in a Lambda function written using the regular programming model. Code written using the framework uses simpler expressions that allow you to focus on your business logic.

The following example code shows how using the annotations framework can simplify writing Lambda functions. The first example shows code written using the regular Lambda program model, and the second shows the equivalent using the Annotations framework.

```csharp
public APIGatewayHttpApiV2ProxyResponse LambdaMathAdd(APIGatewayHttpApiV2ProxyRequest request, ILambdaContext context)
{
    if (!request.PathParameters.TryGetValue("x", out var xs))
    {
        return new APIGatewayHttpApiV2ProxyResponse
```
```csharp
{
    StatusCode = (int)HttpStatusCode.BadRequest
};
}
if (!request.PathParameters.TryGetValue("y", out var ys))
{
    return new APIGatewayHttpApiV2ProxyResponse
    {
        StatusCode = (int)HttpStatusCode.BadRequest
    };
}
var x = int.Parse(xs);
var y = int.Parse(ys);
return new APIGatewayHttpApiV2ProxyResponse
{
    StatusCode = (int)HttpStatusCode.OK,
    Body = (x + y).ToString(),
    Headers = new Dictionary<string, string> { { "Content-Type", "text/plain" } }
};
}

[LambdaFunction]
[HttpApi(LambdaHttpMethod.Get, "/add/{x}/{y}")]
public int Add(int x, int y)
{
    return x + y;
}
```

For another example of how using Lambda Annotations can simplify your code, see this cross-service example application in the awsdocs/aws-doc-sdk-examples GitHub repository. The folder PamApiAnnotations uses Lambda Annotations in the main function.cs file. For comparison, the PamApi folder has equivalent files written using the regular Lambda programming model.

The Annotations framework uses source generators to generate code that translates from the Lambda programming model to the code seen in the second example.

For more information about how to use Lambda Annotations for .NET, see the following resources:

- The [aws/aws-lambda-dotnet](https://github.com/aws/aws-lambda-dotnet) GitHub repository.
• The Amazon.Lambda.Annotations NuGet package.

**Dependency injection with Lambda Annotations framework**

You can also use the Lambda Annotations framework to add dependency injection to your Lambda functions using syntax you are familiar with. When you add a [LambdaStartup] attribute to a Startup.cs file, the Lambda Annotations framework will generate the required code at compile time.

```
[LambdaStartup]
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IDatabaseRepository, DatabaseRepository>();
    }
}
```

Your Lambda function can inject services using either constructor injection or by injecting into individual methods using the [FromServices] attribute.

```
[assembly:
]
namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function(IDatabaseRepository repo)
    {
        this._repo = repo;
    }

    [LambdaFunction]
    [HttpApi(LambdaHttpMethod.Get, "/product/{id}"")]
    public async Task<Product> FunctionHandler([FromServices] IDatabaseRepository repository, string id)
    {
```
Lambda function handler restrictions

Note that there are some restrictions on the handler signature.

- It may not be unsafe and use pointer types in the handler signature, though you can use unsafe context inside the handler method and its dependencies. For more information, see unsafe (C# Reference) on the Microsoft Docs website.
- It may not pass a variable number of parameters using the params keyword, or use ArgIterator as an input or a return parameter, which is used to support a variable number of parameters.
- The handler may not be a generic method, for example, IList<T> Sort<T>(IList<T> input).
- Async handlers with signature async void are not supported.
Build and deploy C# Lambda functions with .zip file archives

A .NET deployment package (.zip file archive) contains your function's compiled assembly along with all of its assembly dependencies. The package also contains a \texttt{proj.deps.json} file. This signals to the .NET runtime all of your function's dependencies and a \texttt{proj.runtimeconfig.json} file, which is used to configure the runtime.

To deploy individual Lambda functions, you can use the Amazon.Lambda.Tools .NET Lambda Global CLI. Using the \texttt{dotnet lambda deploy-function} command automatically creates a .zip deployment package and deploys it to Lambda. However, we recommend that you use frameworks like the AWS Serverless Application Model (AWS SAM) or the AWS Cloud Development Kit (AWS CDK) to deploy your .NET applications to AWS.

Serverless applications usually comprise a combination of Lambda functions and other managed AWS services working together to perform a particular business task. AWS SAM and AWS CDK simplify building and deploying Lambda functions with other AWS services at scale. The AWS SAM \texttt{template specification} provides a simple and clean syntax to describe Lambda functions, APIs, permissions, configurations, and other AWS resources that make up your serverless application. With the AWS CDK you define cloud infrastructure as code to help you build reliable, scalable, cost-effective applications in the cloud using modern programming languages and frameworks like .NET. Both the AWS CDK and the AWS SAM use the .NET Lambda Global CLI to package your functions.

While it's possible to use Lambda layers with functions in C# by using the .NET Core CLI, we recommend against it. Functions in C# that use layers manually load the shared assemblies into memory during the \texttt{Init phase}, which can increase cold start times. Instead, include all shared code at compile time to take advantage of the built-in optimizations of the .NET compiler.

You can find instructions for building and deploying .NET Lambda functions using the AWS SAM, the AWS CDK, and the .NET Lambda Global CLI in the following sections.

Topics

- Using the .NET Lambda Global CLI
- Using the AWS Serverless Application Model (AWS SAM)
- Using the AWS Cloud Development Kit (AWS CDK)
- Deploy ASP.NET applications
Using the .NET Lambda Global CLI

The .NET CLI and the .NET Lambda Global Tools extension (Amazon.Lambda.Tools) offer a cross-platform way to create .NET-based Lambda applications, package them, and deploy them to Lambda. In this section, you learn how to create new Lambda .NET projects using the .NET CLI and Amazon Lambda templates, and to package and deploy them using Amazon.Lambda.Tools

Topics

- Prerequisites
- Creating .NET projects using the .NET CLI
- Deploying .NET projects using the .NET CLI
- Using Lambda layers with the .NET CLI

Prerequisites

.NET 6 SDK

If you haven't already done so, install the .NET 6 SDK and Runtime.

AWS Amazon.Lambda.Templates .NET project templates

To generate your Lambda function code, use the Amazon.Lambda.Templates NuGet package. To install this template package, run the following command:

```
dotnet new --install Amazon.Lambda.Templates
```

AWS Amazon.Lambda.Tools .NET Global CLI tools

To create your Lambda functions, you use the Amazon.Lambda.Tools .NET Global Tools extension. To install Amazon.Lambda.Tools, run the following command:

```
dotnet tool install -g Amazon.Lambda.Tools
```

For more information about the Amazon.Lambda.Tools .NET CLI extension, see the AWS Extensions for .NET CLI repository on GitHub.
Creating .NET projects using the .NET CLI

In the .NET CLI, you use the `dotnet new` command to create .NET projects from the command line. Lambda offers additional templates using the `Amazon.Lambda.Templates` NuGet package.

After installing this package, run the following command to see a list of the available templates.

```shell
dotnet new list
```

To examine details about a template, use the `help` option. For example, to see details about the `lambda.EmptyFunction` template, run the following command.

```shell
dotnet new lambda.EmptyFunction --help
```

To create a basic template for a .NET Lambda function, use the `lambda.EmptyFunction` template. This creates a simple function that takes a string as input and converts it to upper case using the `ToUpper` method. This template supports the following options:

- `--name` – The name of the function.
- `--region` – The AWS Region to create the function in.
- `--profile` – The name of a profile in your AWS SDK for .NET credentials file. To learn more about credential profiles in .NET, see Configure AWS credentials in the AWS SDK for .NET Developer Guide.

In this example, we create a new empty function named `myDotnetFunction` using the default profile and AWS Region settings:

```shell
dotnet new lambda.EmptyFunction --name myDotnetFunction
```

This command creates the following files and directories in your project directory.

```mermaid
### myDotnetFunction
### src
#   ### myDotnetFunction
#     ### Function.cs
#     ### Readme.md
#     ### aws-lambda-tools-defaults.json
#     ### myDotnetFunction.csproj
```
Under the src/myDotnetFunction directory, examine the following files:

- **aws-lambda-tools-defaults.json**: This is where you specify the command line options when deploying your Lambda function. For example:

  ```json
  "profile" : "default",
  "region" : "us-east-2",
  "configuration" : "Release",
  "function-architecture": "x86_64",
  "function-runtime": "dotnet6",
  "function-memory-size" : 256,
  "function-timeout" : 30,
  "function-handler" : "myDotnetFunction::myDotnetFunction.Function::FunctionHandler"
  ```

- **Function.cs**: Your Lambda handler function code. It’s a C# template that includes the default Amazon.Lambda.Core library and a default LambdaSerializer attribute. For more information on serialization requirements and options, see [Serialization in Lambda functions](#). It also includes a sample function that you can edit to apply your Lambda function code.

```csharp
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer))]

namespace myDotnetFunction;

class Function
{
    /// <summary>
    /// A simple function that takes a string and does a ToUpper
    /// </summary#
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
```
public string FunctionHandler(string input, ILambdaContext context)
{
    return input.ToUpper();
}

- **myDotnetFunction.csproj**: An [MSBuild](https://docs.microsoft.com/en-us/dotnet/msbuild/what-is-msbuild) file that lists the files and assemblies that comprise your application.

```xml
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <TargetFramework>net6.0</TargetFramework>
        <ImplicitUsings>enable</ImplicitUsings>
        <Nullable>enable</Nullable>
        <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
        <AWSProjectType>Lambda</AWSProjectType>
        <!-- Makes the build directory similar to a publish directory and helps the AWS .NET Lambda Mock Test Tool find project dependencies. -->
        <CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>
        <!-- Generate ready to run images during publishing to improve cold start time. -->
        <PublishReadyToRun>true</PublishReadyToRun>
    </PropertyGroup>
    <ItemGroup>
        <PackageReference Include="Amazon.Lambda.Core" Version="2.1.0" />
        <PackageReference Include="Amazon.Lambda.Serialization.SystemTextJson" Version="2.3.1" />
    </ItemGroup>
</Project>
```

- **Readme**: Use this file to document your Lambda function.

Under the myfunction/test directory, examine the following files:

- **myFunction.Tests.csproj**: As noted previously, this is an [MSBuild](https://docs.microsoft.com/en-us/dotnet/msbuild/what-is-msbuild) file that lists the files and assemblies that comprise your test project. Note also that it includes the Amazon.Lambda.Core library, so you can seamlessly integrate any Lambda templates required to test your function.

```xml
<Project Sdk="Microsoft.NET.Sdk">
```
• **FunctionTest.cs**: The same C# code template file that it is included in the src directory. Edit this file to mirror your function's production code and test it before uploading your Lambda function to a production environment.

```
using Xunit;
using Amazon.Lambda.Core;
using Amazon.Lambda.TestUtilities;
using MyFunction;

namespace MyFunction.Tests
{
    public class FunctionTest
    {
        [Fact]
        public void TestToUpperFunction()
        {

            // Invoke the lambda function and confirm the string was upper cased.
            var function = new Function();
            var context = new TestLambdaContext();
            var upperCase = function.FunctionHandler("hello world", context);

            Assert.Equal("HELLO WORLD", upperCase);
        }
    }
}
```

**Deploying .NET projects using the .NET CLI**

To build your deployment package and deploy it to Lambda, you use the `Amazon.Lambda.Tools` CLI tools. To deploy your function from the files you created in the previous steps, first navigate into the folder containing your function's .csproj file.

```
cd myDotnetFunction/src/myDotnetFunction
```
To deploy your code to Lambda as a .zip deployment package, run the following command. Choose your own function name.

```bash
dotnet lambda deploy-function myDotnetFunction
```

During the deployment, the wizard asks you to select an section called “Execution role”. For this example, select the lambda_basic_role.

After you have deployed your function, you can test it in the cloud using the `dotnet lambda invoke-function` command. For the example code in the lambda.EmptyFunction template, you can test your function by passing in a string using the `--payload` option.

```bash
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything is OK"
```

If your function has been successfully deployed, you should see output similar to the following.

```bash
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything is OK"
Payload:
"JUST CHECKING IF EVERYTHING IS OK"

Log Tail:
START RequestId: id Version: $LATEST
END RequestId: id
REPORT RequestId: id Duration: 0.99 ms Billed Duration: 1 ms Memory
Size: 256 MB Max Memory Used: 12 MB
```

**Using Lambda layers with the .NET CLI**

**Note**

Using layers with functions in a compiled language like C# may not provide the same amount of benefit as with an interpreted language like Python. Since C# is a compiled language, your functions still have to manually load any shared assemblies into memory during the init phase, which can increase cold start times. Instead, we recommend including any shared code at compile time to take advantage of any built-in compiler optimizations.
The .NET CLI supports commands to help you publish layers and deploy C# functions that consume layers. To publish a layer to a specified Amazon S3 bucket, run the following command in the same directory as your .csproj file:

```
dotnet lambda publish-layer <layer_name> --layer-type runtime-package-store --s3-bucket <s3_bucket_name>
```

Then, when you deploy your function using the .NET CLI, specify the layer ARN the consume in the following command:

```
dotnet lambda deploy-function <function_name> --function-layers arn:aws:lambda:us-east-1:123456789012:layer:layer-name:1
```

For a complete example of a Hello World function, see the [blank-csharp-with-layer](#) sample.

### Using the AWS Serverless Application Model (AWS SAM)

The AWS Serverless Application Model (AWS SAM) is a toolkit that helps streamline the process of building and running serverless applications on AWS. You define the resources for your application in a YAML or JSON template and use the AWS SAM command line interface (AWS SAM CLI) to build, package, and deploy your applications. When you build a Lambda function from an AWS SAM template, AWS SAM automatically creates a .zip deployment package or container image with your function code and any dependencies you specify. AWS SAM then deploys your function using an AWS CloudFormation stack. To learn more about using AWS SAM to build and deploy Lambda functions, see [Getting started with AWS SAM](#) in the AWS Serverless Application Model Developer Guide.

The following steps show you how to download, build, and deploy a sample .NET Hello World application using AWS SAM. This sample application uses a Lambda function and an Amazon API Gateway endpoint to implement a basic API backend. When you send an HTTP GET request to your API Gateway endpoint, API Gateway invokes your Lambda function. The function returns a "hello world" message, along with the IP address of the Lambda function instance that processes your request.

When you build and deploy your application using AWS SAM, behind the scenes the AWS SAM CLI uses the dotnet lambda package command to package the individual Lambda function code bundles.
Prerequisites

.NET 6 SDK

Install the .NET 6 SDK and Runtime.

AWS SAM CLI version 1.39 or later

To learn how to install the latest version of the AWS SAM CLI, see Installing the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello world .NET template using the following command.

```
    sam init --app-template hello-world --name sam-app \
    --package-type Zip --runtime dotnet6
```

This command creates the following files and directories in your project directory.

```text
### sam-app
  ### README.md
  ### events
  #   ### event.json
  ### omnisharp.json
  ### samconfig.toml
  ### src
  #   ### HelloWorld
  #     ### Function.cs
  #     ### HelloWorld.csproj
  #     ### aws-lambda-tools-defaults.json
  ### template.yaml
  ### test
    ### HelloWorld.Test
      ### FunctionTest.cs
      ### HelloWorld.Tests.csproj
```

2. Navigate into the directory containing the template.yaml file. This file is a template that defines the AWS resources for your application, including your Lambda function and an API Gateway API.

```
    cd sam-app
```
3. To build the source of your application, run the following command.

   `sam build`

4. To deploy your application to AWS, run the following command.

   `sam deploy --guided`

This command packages and deploys your application with the following series of prompts. To accept the default options, press Enter.

   - **Stack Name**: The name of the stack to deploy to AWS CloudFormation. This name must be unique to your AWS account and AWS Region.
   - **AWS Region**: The AWS Region you want to deploy your app to.
   - **Confirm changes before deploy**: Select yes to manually review any change sets before AWS SAM deploys application changes. If you select no, the AWS SAM CLI automatically deploys application changes.
   - **Allow SAM CLI IAM role creation**: Many AWS SAM templates, including the Hello world one in this example, create AWS Identity and Access Management (IAM) roles to give your Lambda functions permission to access other AWS services. Select Yes to provide permission to deploy a AWS CloudFormation stack that creates or modifies IAM roles.
   - **Disable rollback**: By default, if AWS SAM encounters an error during creation or deployment of your stack, it rolls the stack back to the previous version. Select No to accept this default.
   - **HelloWorldFunction may not have authorization defined, is this okay**: Enter y.
   - **Save arguments to samconfig.toml**: Select yes to save your configuration choices. In the future, you can re-run `sam deploy` without parameters to deploy changes to your application.

5. When the deployment of your application is complete, the CLI returns the Amazon Resource Name (ARN) of the Hello World Lambda function and the IAM role created for it. It also
AWS Lambda

Developer Guide

displays the endpoint of your API Gateway API. To test your application, open the endpoint in
a browser. You should see a response similar to the following.
{"message":"hello world","location":"34.244.135.203"}

6.

To delete your resources, run the following command. Note that the API endpoint you created
is a public endpoint accessible over the internet. We recommend that you delete this endpoint
after testing.
sam delete

Next steps
To learn more about using AWS SAM to build and deploy Lambda functions using .NET, see the
following resources:
• The AWS Serverless Application Model (AWS SAM) Developer Guide
• Building Serverless .NET Applications with AWS Lambda and the SAM CLI

Using the AWS Cloud Development Kit (AWS CDK)
The AWS Cloud Development Kit (AWS CDK) is an open-source software development framework
for deﬁning cloud infrastructure as code with modern programming languages and frameworks
like .NET. AWS CDK projects are executed to generate AWS CloudFormation templates which are
then used to deploy your code.
To build and deploy an example Hello world .NET application using the AWS CDK, follow the
instructions in the following sections. The sample application implements a basic API backend
consisting of an API Gateway endpoint and a Lambda function. API Gateway invokes the Lambda
function when you send an HTTP GET request to the endpoint. The function returns a Hello world
message, along with the IP address of the Lambda instance that processes your request.

Prerequisites
.NET 6 SDK
Install the .NET 6 SDK and Runtime.
Using the AWS Cloud Development Kit (AWS CDK)

870


Deploy a sample AWS CDK application

1. Create a project directory for the sample application and navigate into it.

   ```
   mkdir hello-world
   cd hello-world
   ```

2. Initialize a new AWS CDK application by running the following command.

   ```
   cdk init app --language csharp
   ```

   The command creates the following files and directories in your project directory:

   ```
   ### README.md
   ### cdk.json
   ### src
   ### HelloWorld
   ### GlobalSuppressions.cs
   ### HelloWorld.csproj
   ### HelloWorldStack.cs
   ### Program.cs
   ### HelloWorld.sln
   ```

3. Open the `src` directory and create a new Lambda function using the .NET CLI. This is the function you will deploy using the AWS CDK. In this example, you create a Hello world function named `HelloWorldLambda` using the `lambda.EmptyFunction` template.

   ```
   cd src
   dotnet new lambda.EmptyFunction -n HelloWorldLambda
   ```

   After this step, your directory structure inside your project directory should look like the following:

   ```
   ### README.md
   ### cdk.json
   ```
4. Open the `HelloWorldStack.cs` file from the `src/HelloWorld` directory. Replace the contents of the file with the following code.

```csharp
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.Logs;
using Constructs;
namespace CdkTest
{
    public class HelloWorldStack : Stack
    {
        internal HelloWorldStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props)
        {
            var buildOption = new BundlingOptions()
            {
                Image = Runtime.DOTNET_6.BundlingImage,
                User = "root",
                OutputType = BundlingOutput.ARCHIVED,
                Command = new string[]
                {
                    "/bin/sh",
                    "-c",
                    " dotnet tool install -g Amazon.Lambda.Tools"+
```
This is the code to compile and bundle the application code, as well as the definition of the Lambda function itself. The BundlingOptions object allows a zip file to be created, along with a set of commands that are used to generate the contents of the zip file. In this instance, the dotnet lambda package command is used to compile and generate the zip file.

5. To deploy your application, run the following command.

   cdk deploy

6. Invoke your deployed Lambda function using the .NET Lambda CLI.

   dotnet lambda invoke-function HelloWorldFunction -p "hello world"

7. After you’ve finished testing, you can delete the resources you created, unless you want to retain them. Run the following command to delete your resources.

   cdk destroy
Next steps

To learn more about using AWS CDK to build and deploy Lambda functions using .NET, see the following resources:

- Working with the AWS CDK in C#
- Build, package, and publish .NET C# Lambda functions with the AWS CDK

Deploy ASP.NET applications

As well as hosting event-driven functions, you can also use .NET with Lambda to host lightweight ASP.NET applications. You can build and deploy ASP.NET applications using the Amazon.Lambda.AspNetCoreServer NuGet package. In this section, you learn how to deploy an ASP.NET web API to Lambda using the .NET Lambda CLI tooling.

Topics

- Prerequisites
- Deploying an ASP.NET Web API to Lambda
- Deploying ASP.NET minimal APIs to Lambda

Prerequisites

.NET 6 SDK

Install the .NET 6 SDK and ASP.NET Core Runtime.

Amazon.Lambda.Tools

To create your Lambda functions, you use the Amazon.Lambda.Tools .NET Global Tools extension. To install Amazon.Lambda.Tools, run the following command:

```
dotnet tool install -g Amazon.Lambda.Tools
```

For more information about the Amazon.Lambda.Tools .NET CLI extension, see the AWS Extensions for .NET CLI repository on GitHub.

Amazon.Lambda.Templates

To generate your Lambda function code, use the Amazon.Lambda.Templates NuGet package. To install this template package, run the following command:
dotnet new --install Amazon.Lambda.Templates

Deploying an ASP.NET Web API to Lambda

To deploy a web API using ASP.NET, you can use the .NET Lambda templates to create a new web API project. Use the following command to initialize a new ASP.NET web API project. In the example command, we name the project AspNetOnLambda.

dotnet new serverless.AspNetCoreWebAPI -n AspNetOnLambda

This command creates the following files and directories in your project directory.

```
.
### AspNetOnLambda
### src
#   ### AspNetOnLambda
#   #   ### AspNetOnLambda.csproj
#   #   ### Controllers
#   # #   ### ValuesController.cs
#   #   ### LambdaEntryPoint.cs
#   #   ### LocalEntryPoint.cs
#   # #   ### Readme.md
#   #   ### Startup.cs
#   #   ### appsettings.Development.json
#   #   ### appsettings.json
#   # #   ### aws-lambda-tools-defaults.json
#   #   ### serverless.template
### test
#   ### AspNetOnLambda.Tests
#   #   ### AspNetOnLambda.Tests.csproj
#   #   ### SampleRequests
# #   ### ValuesController-Get.json
#   #   ### ValuesControllerTests.cs
#   #   ### appsettings.json
```

When Lambda invokes your function, the entry point it uses is the LambdaEntryPoint.cs file. The file created by the .NET Lambda template contains the following code.

```
namespace AspNetOnLambda;
```
public class LambdaEntryPoint : Amazon.Lambda.AspNetCoreServer.APIGatewayProxyFunction
{
    protected override void Init(IWebHostBuilder builder)
    {
        builder
            .UseStartup<Startup>();
    }
}

protected override void Init(IHostBuilder builder)
{
}

The entry point used by Lambda must inherit from one of the three base classes in the
Amazon.Lambda.AspNetCoreServer package. These three base classes are:

- APIGatewayProxyFunction
- APIGatewayHttpApiV2ProxyFunction
- ApplicationLoadBalancerFunction

The default class used when you create your LambdaEntryPoint.cs file using the provided .NET Lambda template is APIGatewayProxyFunction. The base class you use in your function depends on which API layer sits in front of your Lambda function.

Each of the three base classes contains a public method named FunctionHandlerAsync. The name of this method will form part of the handler string Lambda uses to invoke your function. The FunctionHandlerAsync method transforms the inbound event payload into the correct ASP.NET format and the ASP.NET response back to a Lambda response payload. For the example AspNetOnLambda project shown, the handler string would be as follows.

AspNetOnLambda::AspNetOnLambda.LambdaEntryPoint::FunctionHandlerAsync

To deploy the API to Lambda, run the following commands to navigate into the directory containing your source code file and deploy your function using AWS CloudFormation.

cd AspNetOnLambda/src/AspNetOnLambda

dotnet lambda deploy-serverless

Deploy ASP.NET applications
Deploying ASP.NET minimal APIs to Lambda

To deploy an ASP.NET minimal API to Lambda, you can use the .NET Lambda templates to create a new minimal API project. Use the following command to initialize a new minimal API project. In this example, we name the project MinimalApiOnLambda.

```
dotnet new serverless.AspNetCoreMinimalAPI -n MinimalApiOnLambda
```

The command creates the following files and directories in your project directory.

```markdown
### MinimalApiOnLambda
### src
### MinimalApiOnLambda
### Controllers
### CalculatorController.cs
### MinimalApiOnLambda.csproj
### Program.cs
### Readme.md
### appsettings.Development.json
### appsettings.json
### aws-lambda-tools-defaults.json
### serverless.template
```

The `Program.cs` file contains the following code.

```csharp
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();

// Add AWS Lambda support. When application is run in Lambda Kestrel is swapped out as the web server with Amazon.Lambda.AspNetCoreServer. This package will act as the webserver translating request and responses between the Lambda event source and ASP.NET Core.
builder.Services.AddAWSLambdaHosting(LambdaEventSource.RestApi);

var app = builder.Build();

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
```
To configure your minimal API to run on Lambda, you may need to edit this code so that requests and responses between Lambda and ASP.NET Core are properly translated. By default, the function is configured for a REST API event source. For an HTTP API or application load balancer, replace (LambdaEventSource.RestApi) with one of the following options:

- (LambdaEventSource.HttpAPI)
- (LambdaEventSource.ApplicationLoadBalancer)

To deploy your minimal API to Lambda, run the following commands to navigate into the directory containing your source code file and deploy your function using AWS CloudFormation.

```bash
cd MinimalApiOnLambda/src/MinimalApiOnLambda
dotnet lambda deploy-serverless
```
Deploy .NET Lambda functions with container images

There are three ways to build a container image for a .NET Lambda function:

- **Using an AWS base image for .NET**

  The [AWS base images](aws.amazon.com/based) are preloaded with a language runtime, a runtime interface client to manage the interaction between Lambda and your function code, and a runtime interface emulator for local testing.

- **Using an AWS OS-only base image**

  [AWS OS-only base images](aws.amazon.com/based) contain an Amazon Linux distribution and the [runtime interface emulator](aws.amazon.com/based). These images are commonly used to create container images for compiled languages, such as [Go](aws.amazon.com/based) and [Rust](aws.amazon.com/based), and for a language or language version that Lambda doesn't provide a base image for, such as Node.js 19. You can also use OS-only base images to implement a custom runtime. To make the image compatible with Lambda, you must include the [runtime interface client](aws.amazon.com/based) for .NET in the image.

- **Using a non-AWS base image**

  You can use an alternative base image from another container registry, such as Alpine Linux or Debian. You can also use a custom image created by your organization. To make the image compatible with Lambda, you must include the [runtime interface client](aws.amazon.com/based) for .NET in the image.

### Tip

To reduce the time it takes for Lambda container functions to become active, see [Use multi-stage builds](aws.amazon.com/based) in the Docker documentation. To build efficient container images, follow the [Best practices for writing Dockerfiles](aws.amazon.com/based).

This page explains how to build, test, and deploy container images for Lambda.

**Topics**

- [AWS base images for .NET](aws.amazon.com/based)
- [Using an AWS base image for .NET](aws.amazon.com/based)
- [Using an alternative base image with the runtime interface client](aws.amazon.com/based)
**AWS base images for .NET**

AWS provides the following base images for .NET:

<table>
<thead>
<tr>
<th>Tags</th>
<th>Runtime</th>
<th>Operating system</th>
<th>Dockerfile</th>
<th>Deprecation</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>.NET 7</td>
<td>Amazon Linux 2</td>
<td><a href="#">Dockerfile for .NET 7 on GitHub</a></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>.NET 6</td>
<td>Amazon Linux 2</td>
<td><a href="#">Dockerfile for .NET 6 on GitHub</a></td>
<td>Nov 12, 2024</td>
</tr>
</tbody>
</table>

Amazon ECR repository: [gallery.ecr.aws/lambda/dotnet](#)

**Using an AWS base image for .NET**

**Prerequisites**

To complete the steps in this section, you must have the following:

- **.NET SDK** – The following steps use the .NET 7 base image. Make sure that your .NET version matches the version of the base image that you specify in your Dockerfile.
- **Docker**

**Creating and deploying an image using a base image**

In the following steps, you use `Amazon.Lambda.Templates` and `Amazon.Lambda.Tools` to create a .NET project. Then, you build a Docker image, upload the image to Amazon ECR, and deploy it to a Lambda function.

1. Install the [Amazon.Lambda.Templates](#) NuGet package.

   ```bash
   dotnet new install Amazon.Lambda.Templates
   ```

2. Create a .NET project using the `lambda.image.EmptyFunction` template.

   ```bash
   dotnet new lambda.image.EmptyFunction --name MyFunction --region us-east-1
   ```
3. Navigate to the `MyFunction/src/MyFunction` directory. This is where the project files are stored. Examine the following files:

   - **aws-lambda-tools-defaults.json** – This file is where you specify the command line options when deploying your Lambda function.
   - **Function.cs** – Your Lambda handler function code. This is a C# template that includes the default Amazon.Lambda.Core library and a default LambdaSerializer attribute. For more information about serialization requirements and options, see [Serialization in Lambda functions](#). You can use the provided code for testing, or replace it with your own.
   - **MyFunction.csproj** – A .NET project file, which lists the files and assemblies that comprise your application.
   - **Readme.md** – This file contains more information about the sample Lambda function.

4. Examine the Dockerfile in the `src/MyFunction` directory. You can use the provided Dockerfile for testing, or replace it with your own. If you use your own, make sure to:

   - Set the **FROM** property to the [URL of the base image](#). Your .NET version must match the version of the base image.
   - Set the **CMD** argument to the Lambda function handler. This should match the image-command in `aws-lambda-tools-defaults.json`.

   **Example Dockerfile**

   ```
   # You can also pull these images from DockerHub amazon/aws-lambda-dotnet:7
   FROM public.ecr.aws/lambda/dotnet:7

   # Copy function code
   COPY publish/* ${LAMBDA_TASK_ROOT}

   # Set the CMD to your handler (could also be done as a parameter override outside of the Dockerfile)
   CMD [ "MyFunction::MyFunction.Function::FunctionHandler" ]
   ```

5. Install the Amazon.Lambda.Tools .NET Global Tool.

   ```
   dotnet tool install -g Amazon.Lambda.Tools
   ```

   If Amazon.Lambda.Tools is already installed, make sure that you have the latest version.
6. Change the directory to `MyFunction/src/MyFunction`, if you're not there already.

   ```
cd src/MyFunction
   ```

7. Use Amazon.Lambda.Tools to build the Docker image, push it to a new Amazon ECR repository, and deploy the Lambda function.

   For `--function-role`, specify the role name—not the Amazon Resource Name (ARN)—of the execution role for the function. For example, `lambda-role`.

   ```
dotnet lambda deploy-function MyFunction --function-role lambda-role
   ```

   For more information about the Amazon.Lambda.Tools .NET Global Tool, see the AWS Extensions for .NET CLI repository on GitHub.

8. Invoke the function.

   ```
dotnet lambda invoke-function MyFunction --payload "Testing the function"
   ```

   If everything is successful, you see the following:

   **Payload:**
   "TESTING THE FUNCTION"

   **Log Tail:**
   ```
   START RequestId: id Version: $LATEST
   END RequestId: id
   REPORT RequestId: id Duration: 0.99 ms Billed Duration: 1 ms Memory
   Size: 256 MB Max Memory Used: 12 MB
   ```

9. Delete the Lambda function.

   ```
dotnet lambda delete-function MyFunction
   ```
Using an alternative base image with the runtime interface client

If you use an OS-only base image or an alternative base image, you must include the runtime interface client in your image. The runtime interface client extends the Lambda runtime API, which manages the interaction between Lambda and your function code.

The following example demonstrates how to build a container image for .NET using a non-AWS base image, and how to add the Amazon.Lambda.RuntimeSupport package, which is the Lambda runtime interface client for .NET. The example Dockerfile uses the Microsoft .NET 7 base image.

Prerequisites

To complete the steps in this section, you must have the following:

- .NET SDK – The following steps use a .NET 7 base image. Make sure that your .NET version matches the version of the base image that you specify in your Dockerfile.
- Docker

Creating and deploying an image using an alternative base image

1. Install the Amazon.Lambda.Templates NuGet package.

   ```
dotnet new install Amazon.Lambda.Templates
   ```

2. Create a .NET project using the lambda.CustomRuntimeFunction template. This template includes the Amazon.Lambda.RuntimeSupport package.

   ```
dotnet new lambda.CustomRuntimeFunction --name MyFunction --region us-east-1
   ```

3. Navigate to the MyFunction/src/MyFunction directory. This is where the project files are stored. Examine the following files:

   - aws-lambda-tools-defaults.json – This file is where you specify the command line options when deploying your Lambda function.
   - Function.cs – The code contains a class with a Main method that initializes the Amazon.Lambda.RuntimeSupport library as the bootstrap. The Main method is the entry point for the function's process. The Main method wraps the function handler in a wrapper that the bootstrap can work with. For more information, see Using Amazon.Lambda.RuntimeSupport as a class library in the GitHub repository.
4. Open the `aws-lambda-tools-defaults.json` file and Add the following lines:

```json
"package-type": "image",
"docker-host-build-output-dir": ".\bin\Release\lambda-publish"
```

- **package-type**: Defines the deployment package as a container image.
- **docker-host-build-output-dir**: Sets the output directory for the build process.

**Example aws-lambda-tools-defaults.json**

```json
{
   "Information": [
      "This file provides default values for the deployment wizard inside Visual Studio and the AWS Lambda commands added to the .NET Core CLI.",
      "To learn more about the Lambda commands with the .NET Core CLI execute the following command at the command line in the project root directory.",
      "dotnet lambda help",
      "All the command line options for the Lambda command can be specified in this file."
   ],
   "profile": "",
   "region": "us-east-1",
   "configuration": "Release",
   "function-runtime": "provided.al2",
   "function-memory-size": 256,
   "function-timeout": 30,
   "function-handler": "bootstrap",
   "msbuild-parameters": "--self-contained true",
   "package-type": "image",
   "docker-host-build-output-dir": ".\bin\Release\lambda-publish"
}
```

5. Create a Dockerfile in the `MyFunction/src/MyFunction` directory. The following example Dockerfile uses a Microsoft .NET base image instead of an **AWS base image**.

- Set the **FROM** property to the base image identifier. Your .NET version must match the version of the base image.
• Use the COPY command to copy the function into the /var/task directory.

• Set the ENTRYPOINT to the module that you want the Docker container to run when it starts. In this case, the module is the bootstrap, which initializes the Amazon.Lambda.RuntimeSupport library.

Example Dockerfile

```
# You can also pull these images from DockerHub amazon/aws-lambda-dotnet:7
FROM mcr.microsoft.com/dotnet/runtime:7.0

# Set the image's internal work directory
WORKDIR /var/task

# Copy function code
COPY "bin/Release/net7.0/linux-x64" .

# Set the entrypoint to the bootstrap
ENTRYPOINT ["/usr/bin/dotnet", "exec", "/var/task/bootstrap.dll"]
```


```
dotnet tool install -g Amazon.Lambda.Tools
```

If Amazon.Lambda.Tools is already installed, make sure that you have the latest version.

```
dotnet tool update -g Amazon.Lambda.Tools
```

7. Use Amazon.Lambda.Tools to build the Docker image, push it to a new Amazon ECR repository, and deploy the Lambda function.

For --function-role, specify the role name—not the Amazon Resource Name (ARN)—of the execution role for the function. For example, lambda-role.

```
dotnet lambda deploy-function MyFunction --function-role lambda-role
```

For more information about the Amazon.Lambda.Tools .NET CLI extension, see the AWS Extensions for .NET CLI repository on GitHub.

8. Invoke the function.
dotnet lambda invoke-function MyFunction --payload "Testing the function"

If everything is successful, you see the following:

Payload:
"TESTING THE FUNCTION"

Log Tail:
START RequestId: id Version: $LATEST
END RequestId: id
REPORT RequestId: id Duration: 0.99 ms Billed Duration: 1 ms Memory Size: 256 MB Max Memory Used: 12 MB

9. Delete the Lambda function.

dotnet lambda delete-function MyFunction
.NET functions with native AOT compilation

.NET 7 supports native ahead-of-time (AOT) compilation. With native AOT, you can compile your Lambda function code to a native runtime format, which removes the need to compile .NET code at runtime. Native AOT compilation can reduce the cold start time for Lambda functions that you write in .NET. For more information, see Building serverless .NET applications on AWS Lambda using .NET 7 on the AWS Compute Blog.

Sections

• Lambda runtime
• Prerequisites
• Getting started
• Serialization
• Trimming
• Troubleshooting

Lambda runtime

To deploy a Lambda function build with native AOT compilation, use the provided.al2023 or provided.al2 OS-only runtime with the x86_64 architecture. When you use a managed .NET Lambda runtime, your application is compiled into Intermediate Language (IL) code. At runtime, the just-in-time (JIT) compiler in the runtime takes the IL code and compiles it into machine code as needed. With a Lambda function that is compiled ahead of time with native AOT, the runtime environment doesn't include the .NET SDK or .NET runtime. You compile your code into machine code before it runs.

One limitation of AOT is that your application code must be compiled in an environment with the same operating system and processor architecture as the Amazon Linux 2 (AL2) runtime environment. The .NET Lambda CLI provides functionality to compile your application in a Docker container using an AL2 image.

Prerequisites

Docker

To use native AOT, your function code must be compiled in an environment with the same Amazon Linux 2 operating system as the OS-only runtime. The .NET CLI commands in the
following sections use Docker to develop and build Lambda functions in an Amazon Linux 2 environment.

.NET 7 SDK

Native AOT compilation is a feature of .NET 7. You must install the .NET 7 SDK on your build machine, not only the runtime.

Amazon.Lambda.Tools

To create your Lambda functions, you use the Amazon.Lambda.Tools .NET Global Tools extension. To install Amazon.Lambda.Tools, run the following command:

```
dotnet tool install -g Amazon.Lambda.Tools
```

For more information about the Amazon.Lambda.Tools .NET CLI extension, see the AWS Extensions for .NET CLI repository on GitHub.

Amazon.Lambda.Templates

To generate your Lambda function code, use the Amazon.Lambda.Templates NuGet package. To install this template package, run the following command:

```
dotnet new install Amazon.Lambda.Templates
```

Getting started

Both the .NET Global CLI and the AWS Serverless Application Model (AWS SAM) provide getting started templates for building applications using native AOT. To build your first native AOT Lambda function, carry out the steps in the following instructions.

To initialize and deploy a native AOT compiled Lambda function

1. Initialize a new project using the native AOT template and then navigate into the directory containing the created .cs and .csproj files. In this example, we name our function NativeAotSample.

```
dotnet new lambda.NativeAOT -n NativeAotSample
cd ./NativeAotSample/src/NativeAotSample
```
The Function.cs file created by the native AOT template contains the following function code.

```csharp
using Amazon.Lambda.Core;
using Amazon.Lambda.RuntimeSupport;
using Amazon.Lambda.Serialization.SystemTextJson;
using System.Text.Json.Serialization;

namespace NativeAotSample;

public class Function
{
    private static async Task Main()
    {
        Func<string, ILambdaContext, string> handler = FunctionHandler;
        await LambdaBootstrapBuilder.Create(handler, new SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>()).Build().RunAsync();
    }

    public static string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}

[JsonSerializable(typeof(string))]
public partial class LambdaFunctionJsonSerializerContext : JsonSerializerContext
{
}
```

Native AOT compiles your application into a single, native binary. The entrypoint of that binary is the static `Main` method. Within static `Main`, the Lambda runtime is bootstrapped and the `FunctionHandler` method set up. As part of the runtime bootstrap, a source generated serializer is configured using `new SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>()`.

2. To deploy your application to Lambda, ensure that Docker is running in your local environment and run the following command.
dotnet lambda deploy-function

Behind the scenes, the .NET global CLI downloads an Amazon Linux 2 Docker image and compiles your application code inside a running container. The compiled binary is output back to your local filesystem before being deployed to Lambda.

3. Test your function by running the following command. Replace `<FUNCTION_NAME>` with the name you chose for your function in the deployment wizard.

   dotnet lambda invoke-function `<FUNCTION_NAME>` --payload "hello world"

   The response from the CLI includes performance details for the cold start (initialization duration) and total run time for your function invocation.

4. To delete the AWS resources you created by following the preceding steps, run the following command. Replace `<FUNCTION_NAME>` with the name you chose for your function in the deployment wizard. By deleting AWS resources that you're no longer using, you prevent unnecessary charges being billed to your AWS account.

   dotnet lambda delete-function `<FUNCTION_NAME>`

Serialization

To deploy functions to Lambda using native AOT, your function code must use source generated serialization. Instead of using run-time reflection to gather the metadata needed to access object properties for serialization, source generators generate C# source files that are compiled when you build your application. To configure your source generated serializer correctly, ensure that you include any input and output objects your function uses, as well as any custom types. For example, a Lambda function that receives events from an API Gateway REST API and returns a custom `Product` type would include a serializer defined as follows.

```csharp
[JsonSerializable(typeof(APIGatewayProxyRequest))]
[JsonSerializable(typeof(APIGatewayProxyResponse))]
[JsonSerializable(typeof(Product))]
public partial class CustomSerializer : JsonSerializerContext
{
}
```
Trimming

Native AOT trims your application code as part of the compilation to ensure that the binary is as small as possible. Much of the .NET ecosystem is not yet trim friendly. This means that parts of libraries that your function relies on may be trimmed out as part of the compilation step. You can avoid this by defining `TrimmerRootAssemblies` as part of your `.csproj` file.

```xml
<ItemGroup>
  <TrimmerRootAssembly Include="AWSSDK.Core" />
  <TrimmerRootAssembly Include="AWSXRayRecorder.Core" />
  <TrimmerRootAssembly Include="AWSXRayRecorder.Handlers.AwsSdk" />
  <TrimmerRootAssembly Include="Amazon.Lambda.APIGatewayEvents" />
  <TrimmerRootAssembly Include="bootstrap" />
  <TrimmerRootAssembly Include="Shared" />
</ItemGroup>
```

Troubleshooting

**Error: Cross-OS native compilation is not supported.**

Your version of the Amazon.Lambda.Tools .NET Core global tool is out of date. Update to the latest version and try again.

**Docker: image operating system "linux" cannot be used on this platform.**

Docker on your system is configured to use Windows containers. Swap to Linux containers to run the native AOT build environment.

**Unhandled Exception: System.ApplicationException: The serializer NativeAoT.MyCustomJsonSerializerContext is missing a constructor that takes in JsonSerializerOptions object**

If you encounter this error when you invoke your Lambda function, add a runtime directive (`rd.xml`) file to your project, and then redeploy.

For more information about common errors, see the [AWS NativeAOT for .NET](https://github.com/aws/aws-lambda-dotnet-native) repository on GitHub.
AWS Lambda context object in C#

When Lambda runs your function, it passes a context object to the handler. This object provides properties with information about the invocation, function, and execution environment.

Context properties

- **FunctionName** – The name of the Lambda function.
- **FunctionVersion** – The version of the function.
- **InvokedFunctionArn** – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- **MemoryLimitInMB** – The amount of memory that's allocated for the function.
- **AwsRequestId** – The identifier of the invocation request.
- **LogGroupName** – The log group for the function.
- **LogStreamName** – The log stream for the function instance.
- **RemainingTime** (TimeSpan) – The number of milliseconds left before the execution times out.
- **Identity** – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
- **ClientContext** – (mobile apps) Client context that's provided to Lambda by the client application.
- **Logger** The logger object for the function.

You can use information in the ILambdaContext object to output information about your function's invocation for monitoring purposes. The following code provides an example of how to add context information to a structured logging framework. In this example, the function adds AwsRequestId to the log outputs. The function also uses the RemainingTime property to cancel an inflight task if the Lambda function timeout is about to be reached.

```csharp
namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;
```
public Function()
{
    this._repo = new DatabaseRepository();
}

public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest request, ILambdaContext context)
{
    Logger.AppendKey("AwsRequestId", context.AwsRequestId);

    var id = request.PathParameters["id"];  

    using var cts = new CancellationTokenSource();

    try
    {
        cts.CancelAfter(context.RemainingTime.Add(TimeSpan.FromSeconds(-1)));

        var databaseRecord = await this._repo.GetById(id, cts.Token);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
    finally
    {
        cts.Cancel();

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.InternalServerError,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
Lambda function logging in C#

AWS Lambda automatically monitors Lambda functions and sends log entries to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation and other output from your function's code to the log stream. For more information about CloudWatch Logs, see Using Amazon CloudWatch logs with AWS Lambda.

Sections

- Creating a function that returns logs
- Tools and libraries
- Using Powertools for AWS Lambda (.NET) and AWS SAM for structured logging
- Using the Lambda console
- Using the CloudWatch console
- Using the AWS Command Line Interface (AWS CLI)
- Deleting logs

Creating a function that returns logs

To output logs from your function code, you can use methods on the Console class, or any logging library that writes to stdout or stderr.

The .NET runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details.

REPORT line data fields

- RequestId – The unique request ID for the invocation.
- Duration – The amount of time that your function's handler method spent processing the event.
- Billed Duration – The amount of time billed for the invocation.
- Memory Size – The amount of memory allocated to the function.
- Max Memory Used – The amount of memory used by the function.
- Init Duration – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- XRAY TraceId – For traced requests, the AWS X-Ray trace ID.
• **SegmentId** – For traced requests, the X-Ray segment ID.
• **Sampled** – For traced requests, the sampling result.

**Tools and libraries**

**Powertools for AWS Lambda (.NET)** is a developer toolkit to implement Serverless best practices and increase developer velocity. The **Logging utility** provides a Lambda optimized logger which includes additional information about function context across all your functions with output structured as JSON. Use this utility to do the following:

• Capture key fields from the Lambda context, cold start and structures logging output as JSON
• Log Lambda invocation events when instructed (disabled by default)
• Print all the logs only for a percentage of invocations via log sampling (disabled by default)
• Append additional keys to structured log at any point in time
• Use a custom log formatter (Bring Your Own Formatter) to output logs in a structure compatible with your organization's Logging RFC

**Using Powertools for AWS Lambda (.NET) and AWS SAM for structured logging**

Follow the steps below to download, build, and deploy a sample Hello World C# application with integrated **Powertools for AWS Lambda (.NET)** modules using the **AWS SAM**. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

**Prerequisites**

To complete the steps in this section, you must have the following:

• .NET 6
• **AWS CLI version 2**
• **AWS SAM CLI version 1.75 or later**. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.
Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.

   ```bash
   sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type Zip --runtime dotnet6 --no-tracing
   ```

2. Build the app.

   ```bash
   cd sam-app && sam build
   ```

3. Deploy the app.

   ```bash
   sam deploy --guided
   ```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.

   **Note**

   For **HelloWorldFunction may not have authorization defined, Is this okay?**, make sure to enter `y`.

5. Get the URL of the deployed application:

   ```bash
   aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
   ```

6. Invoke the API endpoint:

   ```bash
   curl -X GET <URL_FROM_PREVIOUS_STEP>
   ```

   If successful, you'll see this response:

   ```json
   {
   "message": "hello world"
   }
   ```

7. To get the logs for the function, run `sam logs`. For more information, see [Working with logs](#) in the **AWS Serverless Application Model Developer Guide**.

   ```bash
   sam logs --stack-name sam-app
   ```
The log output looks like this:

```plaintext
2023/02/20/$LATEST]4eaf8445ba7a4a93b999cb17fbbecd8
2023-02-20T14:15:27.980000 INIT_START Runtime Version:
dotnet:6.v13    Runtime Version ARN: arn:aws:lambda:ap-southeast-2::runtime:699f346a05dae24c58c45790bc4089f252bf17dae3997e79b17d939a288aa1ec
2023/02/20/$LATEST]4eaf8445ba7a4a93b999cb17fbbecd8 2023-02-20T14:15:28.229000
START RequestId: bed25b38-d012-42e7-ba28-f272535fb80e Version: $LATEST
2023/02/20/$LATEST]4eaf8445ba7a4a93b999cb17fbbecd8 2023-02-20T14:15:29.259000
2023-02-20T14:15:29.201Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
"_aws":{"Timestamp":1676902528962,"CloudWatchMetrics":[{"Namespace":"sam-app-logging","Metrics":[{"Name":"ColdStart","Unit":"Count"}],"Dimensions": [{"FunctionName":["sam-app-HelloWorldFunction-haKIoVeose2p","Service":"PowertoolsHelloWorld","ColdStart":1}
2023/02/20/$LATEST]4eaf8445ba7a4a93b999cb17fbbecd8 2023-02-20T14:15:30.479000
2023-02-20T14:15:30.479Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
"ColdStart":true,"XrayTraceId":"1-63f3807f-5dcbcb9910c96f50742707542","CorrelationId":"d3da54a9-4d67b2fcd015","FunctionName":"sam-app-HelloWorldFunction-haKIoVeose2p","Service":"PowertoolsHelloWorld","ColdStart":1}
2023/02/20/$LATEST]4eaf8445ba7a4a93b999cb17fbbecd8 2023-02-20T14:15:30.599000
2023-02-20T14:15:30.599Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
{"_aws":{"Timestamp":1676902528922,"CloudWatchMetrics":[{"Namespace":"sam-app-logging","Metrics":[{"Name":"ApiRequestCount","Unit":"Count"}],"Dimensions": [{"FunctionName":["PowertoolsHelloWorld","ApiRequestCount":1}
2023/02/20/$LATEST]4eaf8445ba7a4a93b999cb17fbbecd8 2023-02-20T14:15:30.599000
2023-02-20T14:15:30.599Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
{"_aws":{"Timestamp":1676902528922,"CloudWatchMetrics":[{"Namespace":"sam-app-logging","Metrics":[{"Name":"ApiRequestCount","Unit":"Count"}],"Dimensions": [{"FunctionName":["PowertoolsHelloWorld","ApiRequestCount":1}
```

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```plaintext
sam delete
```

Managing log retention

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which CloudWatch automatically deletes the logs. To set up log retention, add the following to your AWS SAM template:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Using the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded Code editor, you will find logs in the execution results. When you use the console test feature to invoke a function, you'll find Log output in the Details section.

Using the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the Log groups page on the CloudWatch console.
2. Choose the log group for your function (/aws/lambda/your-function-name).
3. Choose a log stream.

Each log stream corresponds to an instance of your function. A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple
concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see Error processor sample application for AWS Lambda.

**Using the AWS Command Line Interface (AWS CLI)**

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- AWS Command Line Interface (AWS CLI) version 2
- AWS CLI – Quick configuration with `aws configure`

You can use the AWS CLI to retrieve logs for an invocation using the `--log-type` command option. The response contains a LogResult field that contains up to 4 KB of base64-encoded logs from the invocation.

**Example retrieve a log ID**

The following example shows how to retrieve a log ID from the LogResult field for a function named `my-function`.

```sh
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```json
{
    "StatusCode": 200,
    "LogResult": "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTEzTGEzNzQyOTc0YzV1NGZiMjEgVmdld3Nh
    "ExecutedVersion": "$LATEST"
}
```

**Example decode the logs**

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```sh
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```json
{
    "StatusCode": 200,
    "LogResult": "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTEzTGEzNzQyOTc0YzV1NGZiMjEgVmdld3Nh
    "ExecutedVersion": "$LATEST"
}
```
```bash
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --decode
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://aws.amazon.com/blogs/aws/aws-cli-supported-global-command-line-options/) in the *AWS Command Line Interface User Guide for Version 2*.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"
ask/lib:/opt/lib
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Duration: 79.67 ms Billed Duration: 80 ms Memory Size: 128 MB Max Memory Used: 73 MB
```

The `base64` utility is available on Linux, macOS, and [Ubuntu on Windows](https://ubuntu.com). macOS users may need to use `base64 -D`.

### Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

```bash
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --payload '{"key": "value"}' out
sed -i' -e 's/"//g' out
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://aws.amazon.com/blogs/aws/aws-cli-supported-global-command-line-options/) in the *AWS Command Line Interface User Guide for Version 2*.
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-name stream1 --limit 5

Example macOS and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
  "StatusCode": 200,
  "ExecutedVersion": "$LATEST"
}
{
  "events": [
  {
    "timestamp": 1559763003171,
    "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST",
    "ingestionTime": 1559763003309
  },
  {
    "timestamp": 1559763003173,
    "message": "2019-06-05T19:30:03.173Z 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
\r ...}
    "ingestionTime": 1559763018353
  },
  {
    "timestamp": 1559763003173,
    "message": "2019-06-05T19:30:03.173Z t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  "key": "value"
}
    "ingestionTime": 1559763018353
  }
}
```
Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which logs are deleted automatically.
AWS Lambda function errors in C#

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

This page describes how to view Lambda function invocation errors for the C# runtime using the Lambda console and the AWS CLI.

Sections

- Syntax
- How it works
- Using the Lambda console
- Using the AWS Command Line Interface (AWS CLI)
- Error handling in other AWS services
- What's next?

Syntax

In the initialization phase, exceptions can be thrown for invalid handler strings, a rule-breaking type or method (see Lambda function handler restrictions), or any other validation method (such as forgetting the serializer attribute and having a POCO as your input or output type). These exceptions are of type LambdaException. For example:

```
{
    "errorType": "LambdaException",
    "errorMessage": "Invalid lambda function handler: 'http://this.is.not.a.valid.handler/'.
    The valid format is 'ASSEMBLY::TYPE::METHOD'.'
}
```

If your constructor throws an exception, the error type is also of type LambdaException, but the exception thrown during construction is provided in the cause property, which is itself a modeled exception object:

```
{
    "errorType": "LambdaException",
    "errorMessage": "An exception was thrown when the constructor for type 'LambdaExceptionTestFunction.ThrowExceptionInConstructor'
```
was invoked. Check inner exception for more details.

"cause": {
  "errorType": "TargetInvocationException",
  "errorMessage": "Exception has been thrown by the target of an invocation."
}

"stackTrace": [
  "at System.RuntimeTypeHandle.CreateInstance(RuntimeType type, Boolean publicOnly, Boolean noCheck, Boolean&canBeCached, RuntimeMethodHandleInternal&ctor, Boolean& bNeedSecurityCheck)",
  "at System.RuntimeType.CreateInstanceSlow(Boolean publicOnly, Boolean skipCheckThis, Boolean fillCache, StackCrawlMark& stackMark)",
  "at System.Activator.CreateInstance(Type type, Boolean nonPublic)",
  "at System.Activator.CreateInstance(Type type)"
},

"cause": {
  "errorType": "ArithmeticException",
  "errorMessage": "Sorry, 2 + 2 = 5"
}

As the example shows, the inner exceptions are always preserved (as the cause property), and can be deeply nested.

Exceptions can also occur during invocation. In this case, the exception type is preserved and the exception is returned directly as the payload and in the CloudWatch logs. For example:

{
  "errorType": "AggregateException",
  "errorMessage": "One or more errors occurred. (An unknown web exception occurred!)",
  "stackTrace": [
    "at System.Threading.Tasks.Task.ThrowIfExceptional(Boolean includeTaskCanceledExceptions)",
    "at System.Threading.Tasks.Task`1.GetResultCore(Boolean waitCompletionNotification)"
  ],
  "cause": {
    "errorType": "UnknownWebException",
    "errorMessage": "An unknown web exception occurred!"
  }
}
"at LambdaDemo107.LambdaEntryPoint.<GetUriResponse>d__1.MoveNext()",
"--- End of stack trace from previous location where exception was thrown ---",
"at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
"at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)",
"at LambdaDemo107.LambdaEntryPoint.<CheckWebsiteStatus>d__0.MoveNext()"
],
"cause": 
{  
"errorType": "WebException",
"errorMessage": "An error occurred while sending the request. SSL peer certificate or SSH remote key was not OK",
"stackTrace": [  
"at System.Net.HttpWebRequest.EndGetResponse(IAsyncResult asyncResult)",
"at System.Threading.Tasks.TaskFactory`1.FromAsyncCoreLogic(IAsyncResult iar, Func`2 endFunction, Action`1 endAction, Task`1 promise, Boolean requiresSynchronization)",
"--- End of stack trace from previous location where exception was thrown ---",
"at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
"at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)",
"at LambdaDemo107.LambdaEntryPoint.<GetUriResponse>d__1.MoveNext()"
],
"cause": {  
"errorType": "HttpRequestException",
"errorMessage": "An error occurred while sending the request."
"stackTrace": [
"at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
"at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)",
"--- End of stack trace from previous location where exception was thrown ---",
"at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
"at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)",
"--- End of stack trace from previous location where exception was thrown ---",
"at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
"at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)",
"at LambdaDemo107.LambdaEntryPoint.<CheckWebsiteStatus>d__0.MoveNext()"
]
The method in which error information is conveyed depends on the invocation type:

- **RequestResponse** invocation type (that is, synchronous execution): In this case, you get the error message back.
  
  For example, if you invoke a Lambda function using the Lambda console, the RequestResponse is always the invocation type and the console displays the error information returned by AWS Lambda in the **Execution result** section of the console.

- **Event** invocation type (that is, asynchronous execution): In this case AWS Lambda does not return anything. Instead, it logs the error information in CloudWatch Logs and CloudWatch metrics.

### How it works

When you invoke a Lambda function, Lambda receives the invocation request and validates the permissions in your execution role, verifies that the event document is a valid JSON document, and checks parameter values.
If the request passes validation, Lambda sends the request to a function instance. The Lambda runtime environment converts the event document into an object, and passes it to your function handler.

If Lambda encounters an error, it returns an exception type, message, and HTTP status code that indicates the cause of the error. The client or service that invoked the Lambda function can handle the error programatically, or pass it along to an end user. The correct error handling behavior depends on the type of application, the audience, and the source of the error.

The following list describes the range of status codes you can receive from Lambda.

2xx

A 2xx series error with a X-Amz-Function-Error header in the response indicates a Lambda runtime or function error. A 2xx series status code indicates that Lambda accepted the request, but instead of an error code, Lambda indicates the error by including the X-Amz-Function-Error header in the response.

4xx

A 4xx series error indicates an error that the invoking client or service can fix by modifying the request, requesting permission, or by retrying the request. 4xx series errors other than 429 generally indicate an error with the request.

5xx

A 5xx series error indicates an issue with Lambda, or an issue with the function's configuration or resources. 5xx series errors can indicate a temporary condition that can be resolved without any action by the user. These issues can't be addressed by the invoking client or service, but a Lambda function's owner may be able to fix the issue.

For a complete list of invocation errors, see InvokeFunction errors.

Using the Lambda console

You can invoke your function on the Lambda console by configuring a test event and viewing the output. The output is captured in the function's execution logs and, when active tracing is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the Functions page of the Lambda console.
2. Choose the function to test, and choose **Test**.
3. Under **Test event**, select **New event**.
4. Select a **Template**.
5. For **Name**, enter a name for the test. In the text entry box, enter the JSON test event.
6. Choose **Save changes**.
7. Choose **Test**.

The Lambda console invokes your function *synchronously* and displays the result. To see the response, logs, and other information, expand the **Details** section.

### Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface (AWS CLI) version 2](#)
- [AWS CLI – Quick configuration with `aws configure`](#)

When you invoke a Lambda function in the AWS CLI, the AWS CLI splits the response into two documents. The AWS CLI response is displayed in your command prompt. If an error has occurred, the response contains a `FunctionError` field. The invocation response or error returned by the function is written to an output file. For example, `output.json` or `output.txt`.

The following `invoke` command example demonstrates how to invoke a function and write the invocation response to an `output.txt` file.

```bash
aws lambda invoke   \\   --function-name my-function   \\   --cli-binary-format raw-in-base64-out   \\   --payload '{"key1": "value1", "key2": "value2", "key3": "value3"}' output.txt
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#) in the *AWS Command Line Interface User Guide for Version 2*. 


You should see the AWS CLI response in your command prompt:

```json
{
   "StatusCode": 200,
   "FunctionError": "Unhandled",
   "ExecutedVersion": "$LATEST"
}
```

You should see the function invocation response in the `output.txt` file. In the same command prompt, you can also view the output in your command prompt using:

```
cat output.txt
```

You should see the invocation response in your command prompt.

Lambda also records up to 256 KB of the error object in the function's logs. For more information, see [Lambda function logging in C#](#).

## Error handling in other AWS services

When another AWS service invokes your function, the service chooses the invocation type and retry behavior. AWS services can invoke your function on a schedule, in response to a lifecycle event on a resource, or to serve a request from a user. Some services invoke functions asynchronously and let Lambda handle errors, while others retry or pass errors back to the user.

For example, API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502 error code. To customize the error response, you must catch errors in your code and format a response in the required format.

We recommend using AWS X-Ray to determine the source of an error and its cause. X-Ray allows you to find out which component encountered an error, and see details about the errors. The following example shows a function error that resulted in a 502 response from API Gateway.

![Diagram showing error handling in AWS services](image-url)
For more information, see [Instrumenting C# code in AWS Lambda](#).

What's next?

- Learn how to show logging events for your Lambda function on the [section called “Logging”](#) page.
Instrumenting C# code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of three SDK libraries:

- **AWS Distro for OpenTelemetry (ADOT)** – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- **AWS X-Ray SDK for .NET** – An SDK for generating and sending trace data to X-Ray.
- **Powertools for AWS Lambda (.NET)** – A developer toolkit to implement Serverless best practices and increase developer velocity.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

⚠️ **Important**

The X-Ray and Powertools for AWS Lambda SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs.

**Sections**

- Using Powertools for AWS Lambda (.NET) and AWS SAM for tracing
- Using the X-Ray SDK to instrument your .NET functions
- Activating tracing with the Lambda console
- Activating tracing with the Lambda API
- Activating tracing with AWS CloudFormation
- Interpreting an X-Ray trace
Using Powertools for AWS Lambda (.NET) and AWS SAM for tracing

Follow the steps below to download, build, and deploy a sample Hello World C# application with integrated Powertools for AWS Lambda (.NET) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- .NET 6
- AWS CLI version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.

   ```bash
   sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type Zip --runtime dotnet6 --no-tracing
   ```

2. Build the app.

   ```bash
   cd sam-app && sam build
   ```

3. Deploy the app.

   ```bash
   sam deploy --guided
   ```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.
Note

For **HelloWorldFunction** may not have authorization defined, Is this okay?, make sure to enter y.

5. Get the URL of the deployed application:

```bash
aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```bash
curl <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```json
{"message":"hello world"}
```

7. To get the traces for the function, run **sam traces**.

```bash
sam traces
```

The trace output looks like this:

```
New XRay Service Graph
  Start time: 2023-02-20 23:05:16+08:00
  End time: 2023-02-20 23:05:16+08:00
  Reference Id: 0 - AWS::Lambda - sam-app-HelloWorldFunction-pNjujb7mEoew - Edges: [1]
  Summary_statistics:
    - total requests: 1
    - ok count(2XX): 1
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 2.814
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-pNjujb7mEoew - Edges: []
  Summary_statistics:
    - total requests: 1
    - ok count(2XX): 1
```

Using Powertools for AWS Lambda (.NET) and AWS SAM for tracing
- error count(4XX): 0
- fault count(5XX): 0
- total response time: 2.429

Reference Id: 2 - (Root) AWS::ApiGateway::Stage - sam-app/Prod - Edges: [0]
Summary_statistics:
- total requests: 1
- ok count(2XX): 1
- error count(4XX): 0
- fault count(5XX): 0
- total response time: 2.839

Reference Id: 3 - client - sam-app/Prod - Edges: [2]
Summary_statistics:
- total requests: 0
- ok count(2XX): 0
- error count(4XX): 0
- fault count(5XX): 0
- total response time: 0

XRay Event [revision 3] at (2023-02-20T23:05:16.521000) with id (1-63f38c2c-270200bf1d292a442c8e8a00) and duration (2.877s)
- 2.839s - sam-app/Prod [HTTP: 200]
- 2.836s - Lambda [HTTP: 200]
- 2.814s - sam-app-HelloWorldFunction-pNjujb7mEoew [HTTP: 200]
- 2.429s - sam-app-HelloWorldFunction-pNjujb7mEoew
  - 0.230s - Initialization
  - 2.389s - Invocation
    - 0.600s - ## FunctionHandler
    - 0.517s - Get Calling IP
  - 0.039s - Overhead

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.
Using the X-Ray SDK to instrument your .NET functions

You can instrument your function code to record metadata and trace downstream calls. To record detail about calls that your function makes to other resources and services, use the AWS X-Ray SDK for .NET. To get the SDK, add the AWSXRayRecorder packages to your project file.

```xml
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <TargetFramework>net6.0</TargetFramework>
        <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
        <AWSProjectType>Lambda</AWSProjectType>
    </PropertyGroup>
    <ItemGroup>
        <PackageReference Include="Amazon.Lambda.Core" Version="2.1.0" />
        <PackageReference Include="Amazon.Lambda.SQSEvents" Version="2.1.0" />
        <PackageReference Include="Amazon.Lambda.Serialization.Json" Version="2.1.0" />
        <PackageReference Include="AWSSDK.Core" Version="3.7.103.24" />
        <PackageReference Include="AWSSDK.Lambda" Version="3.7.104.3" />
        <PackageReference Include="AWSXRayRecorder.Core" Version="2.13.0" />
        <PackageReference Include="AWSXRayRecorder.Handlers.AwsSdk" Version="2.11.0" />
    </ItemGroup>
</Project>
```

There are a range of Nuget packages that provide auto-instrumentation for AWS SDKs, Entity Framework and HTTP requests. To see the complete set of configuration options refer to [AWS X-Ray SDK for .NET](https://docs.aws.amazon.com/xray/latest/dg/SDKs.html) in the AWS X-Ray Developer Guide.

Once you have added the desired Nuget packages, configure auto-instrumentation. Best practice is to perform this configuration outside of your function's handler function. This allows you to take advantage of execution environment re-use to improve the performance of your function. In the following code example, the RegisterXRayForAllServices method is called in the function constructor to add instrumentation for all AWS SDK calls.

```csharp
[assembly:
]
Activating tracing with the Lambda console

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose Configuration and then choose Monitoring and operations tools.
4. Choose Edit.
5. Under X-Ray, toggle on Active tracing.
6. Choose Save.
Activating tracing with the Lambda API

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- `UpdateFunctionConfiguration`
- `GetFunctionConfiguration`
- `CreateFunction`

The following example AWS CLI command enables active tracing on a function named `my-function`.

```bash
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Activating tracing with AWS CloudFormation

To activate tracing on an `AWS::Lambda::Function` resource in an AWS CloudFormation template, use the `TracingConfig` property.

Example `function-inline.yml` – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
```

For an AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` resource, use the `Tracing` property.

Example `template.yml` – Tracing configuration

```
Resources:
```
Interpreting an X-Ray trace

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's execution role. Otherwise, add the AWSXRayDaemonWriteAccess policy to the execution role.

After you've configured active tracing, you can observe specific requests through your application. The X-Ray service graph shows information about your application and all its components. The following example from the error processor sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

Note

You cannot configure the X-Ray sampling rate for your functions.
When using active tracing, Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes for the primary function from the error processor sample application.

The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of `AWS::Lambda` and the other has origin `AWS::Lambda::Function`.

This example expands the function segment to show its three subsegments:

- **Initialization** – Represents time spent loading your function and running **initialization code**. This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.
You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see the AWS X-Ray SDK for .NET in the AWS X-Ray Developer Guide.

**Pricing**

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see AWS X-Ray pricing.
AWS Lambda function testing in C#

See the Testing functions chapter for a complete introduction to techniques and best practices for testing serverless solutions.

Testing serverless functions uses traditional test types and techniques, but you must also consider testing serverless applications as a whole. Cloud-based tests will provide the most accurate measure of quality of both your functions and serverless applications.

A serverless application architecture includes managed services that provide critical application functionality through API calls. For this reason, your development cycle should include automated tests that verify functionality when your function and services interact.

If you do not create cloud-based tests, you could encounter issues due to differences between your local environment and the deployed environment. Your continuous integration process should run tests against a suite of resources provisioned in the cloud before promoting your code to the next deployment environment, such as QA, Staging, or Production.

Continue reading this short guide to learn about testing strategies for serverless applications, or visit the Serverless Test Samples repository to dive in with practical examples, specific to your chosen language and runtime.
For serverless testing, you will still write *unit*, *integration* and *end-to-end* tests.

- **Unit tests** - Tests that run against an isolated block of code. For example, verifying the business logic to calculate the delivery charge given a particular item and destination.
- **Integration tests** - Tests involving two or more components or services that interact, typically in a cloud environment. For example, verifying a function processes events from a queue.
- **End-to-end tests** - Tests that verify behavior across an entire application. For example, ensuring infrastructure is set up correctly and that events flow between services as expected to record a customer's order.

## Testing your serverless applications

You will generally use a mix of approaches to test your serverless application code, including testing in the cloud, testing with mocks, and occasionally testing with emulators.

### Testing in the cloud

Testing in the cloud is valuable for all phases of testing, including unit tests, integration tests, and end-to-end tests. You run tests against code deployed in the cloud and interacting with cloud-based services. This approach provides the most accurate measure of quality of your code.

A convenient way to debug your Lambda function in the cloud is through the console with a test event. A *test event* is a JSON input to your function. If your function does not require input, the event can be an empty JSON document ( `{}` ). The console provides sample events for a variety of service integrations. After creating an event in the console, you can share it with your team to make testing easier and consistent.

⚠️ **Note**

Testing a function in the console is a quick way to get started, but automating your test cycles ensures application quality and development speed.

### Testing tools

To accelerate your development cycle, there are a number of tools and techniques you can use when testing your functions. For example, [AWS SAM Accelerate](https://aws.amazon.com/sam/) and [AWS CDK watch mode](https://aws.amazon.com/cdk/) both decrease the time required to update cloud environments.
The way you define your Lambda function code makes it simple to add unit tests. Lambda requires a public, parameterless constructor to initialize your class. Introducing a second, internal constructor gives you control of the dependencies your application uses.

```csharp
[assembly:
]
namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function(): this(null)
    {
    }

    internal Function(IDatabaseRepository repo)
    {
        this._repo = repo ?? new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest request)
    {
        var id = request.PathParameters["id"];  

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

To write a test for this function, you can initialize a new instance of your Function class and pass in a mocked implementation of the IDatabaseRepository. The below examples uses Xunit, Moq, and FluentAssertions to write a simple test ensuring the FunctionHandler returns a 200 status code.
using Xunit;
using Moq;
using FluentAssertions;

class FunctionTests
{
    [Fact]
    public async Task TestLambdaHandler_WhenInputIsValid_ShouldReturn200StatusCode()
    {
        // Arrange
        var mockDatabaseRepository = new Mock<IDatabaseRepository>();
        var functionUnderTest = new Function(mockDatabaseRepository.Object);

        // Act
        var response = await functionUnderTest.FunctionHandler(new APIGatewayProxyRequest());

        // Assert
        response.StatusCode.Should().Be(200);
    }
}

For more detailed examples, including examples of asynchronous tests, see the .NET testing samples repository on GitHub.
Building Lambda functions with PowerShell

The following sections explain how common programming patterns and core concepts apply when you author Lambda function code in PowerShell.

Lambda provides the following sample applications for PowerShell:

- **blank-powershell** – A PowerShell function that shows the use of logging, environment variables, and the AWS SDK.

Before you get started, you must first set up a PowerShell development environment. For instructions on how to do this, see [Setting Up a PowerShell Development Environment](#).

To learn about how to use the AWSLambdaPSCore module to download sample PowerShell projects from templates, create PowerShell deployment packages, and deploy PowerShell functions to the AWS Cloud, see [Deploy PowerShell Lambda functions with .zip file archives](#).

Lambda provides the following runtimes for .NET languages:

### .NET

<table>
<thead>
<tr>
<th>Name</th>
<th>Identifier</th>
<th>Operating system</th>
<th>Deprecation date</th>
<th>Block function create</th>
<th>Block function update</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NET 7 (container only)</td>
<td>dotnet7</td>
<td>Amazon Linux 2</td>
<td></td>
<td>Jan 11, 2024</td>
<td>Feb 11, 2024</td>
</tr>
<tr>
<td>.NET 6</td>
<td>dotnet6</td>
<td>Amazon Linux 2</td>
<td>Nov 12, 2024</td>
<td>Jan 11, 2024</td>
<td>Feb 11, 2024</td>
</tr>
</tbody>
</table>

**Topics**

- [Setting Up a PowerShell Development Environment](#)
- [Deploy PowerShell Lambda functions with .zip file archives](#)
- [AWS Lambda function handler in PowerShell](#)
- [AWS Lambda context object in PowerShell](#)
- AWS Lambda function logging in PowerShell
- AWS Lambda function errors in PowerShell
Setting Up a PowerShell Development Environment

Lambda provides a set of tools and libraries for the PowerShell runtime. For installation instructions, see [Lambda tools for PowerShell](https://github.com/aws-lambda-tools-for-powershell) on GitHub.

The AWSLambdaPSCore module includes the following cmdlets to help author and publish PowerShell Lambda functions:

- **Get-AWSPowerShellLambdaTemplate** – Returns a list of getting started templates.
- **New-AWSPowerShellLambda** – Creates an initial PowerShell script based on a template.
- **Publish-AWSPowerShellLambda** – Publishes a given PowerShell script to Lambda.
- **New-AWSPowerShellLambdaPackage** – Creates a Lambda deployment package that you can use in a CI/CD system for deployment.
Deploy PowerShell Lambda functions with .zip file archives

A deployment package for the PowerShell runtime contains your PowerShell script, PowerShell modules that are required for your PowerShell script, and the assemblies needed to host PowerShell Core.

Creating the Lambda function

To get started writing and invoking a PowerShell script with Lambda, you can use the New-AWSPowerShellLambda cmdlet to create a starter script based on a template. You can use the Publish-AWSPowerShellLambda cmdlet to deploy your script to Lambda. Then you can test your script either through the command line or the Lambda console.

To create a new PowerShell script, upload it, and test it, do the following:

1. To view the list of available templates, run the following command:

```powershell
PS C:\> Get-AWSPowerShellLambdaTemplate

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>Bare bones script</td>
</tr>
<tr>
<td>CodeCommitTrigger</td>
<td>Script to process AWS CodeCommit Triggers</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
```

2. To create a sample script based on the Basic template, run the following command:

```powershell
New-AWSPowerShellLambda -ScriptName MyFirstPSScript -Template Basic
```

A new file named MyFirstPSScript.ps1 is created in a new subdirectory of the current directory. The name of the directory is based on the -ScriptName parameter. You can use the -Directory parameter to choose an alternative directory.

You can see that the new file has the following contents:

```powershell
# PowerShell script file to run as a Lambda function
#
# When executing in Lambda the following variables are predefined.
# $LambdaInput - A PSObject that contains the Lambda function input data.
# $LambdaContext - An Amazon.Lambda.Core.ILambdaContext object that contains information about the currently running Lambda environment.
```
# The last item in the PowerShell pipeline is returned as the result of the Lambda function.

To include PowerShell modules with your Lambda function, like the AWSPowerShell.NetCore module, add a "#Requires" statement indicating the module and version.

```powershell
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
```

# Uncomment to send the input to CloudWatch Logs
```powershell
# Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
```

3. To see how log messages from your PowerShell script are sent to Amazon CloudWatch Logs, uncomment the `Write-Host` line of the sample script.

To demonstrate how you can return data back from your Lambda functions, add a new line at the end of the script with `$PSVersionTable`. This adds the `$PSVersionTable` to the PowerShell pipeline. After the PowerShell script is complete, the last object in the PowerShell pipeline is the return data for the Lambda function. `$PSVersionTable` is a PowerShell global variable that also provides information about the running environment.

After making these changes, the last two lines of the sample script look like this:

```powershell
Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
$PSVersionTable
```

4. After editing the `MyFirstPSScript.ps1` file, change the directory to the script's location. Then run the following command to publish the script to Lambda:

```powershell
Publish-AWSPowerShellLambda -ScriptPath .\MyFirstPSScript.ps1 -Name MyFirstPSScript -Region us-east-2
```

Note that the `--Name` parameter specifies the Lambda function name, which appears in the Lambda console. You can use this function to invoke your script manually.

5. Invoke your function using the AWS Command Line Interface (AWS CLI) `invoke` command.

```bash
> aws lambda invoke --function-name MyFirstPSScript out
```
AWS Lambda function handler in PowerShell

When a Lambda function is invoked, the Lambda handler invokes the PowerShell script.

When the PowerShell script is invoked, the following variables are predefined:

- **$LambdaInput** – A PSObject that contains the input to the handler. This input can be event data (published by an event source) or custom input that you provide, such as a string or any custom data object.

- **$LambdaContext** – An Amazon.Lambda.Core.ILambdaContext object that you can use to access information about the current invocation—such as the name of the current function, the memory limit, execution time remaining, and logging.

For example, consider the following PowerShell example code.

```powershell
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function Name:' $LambdaContext.FunctionName
```

This script returns the FunctionName property that's obtained from the $LambdaContext variable.

**Note**

You're required to use the #Requires statement within your PowerShell scripts to indicate the modules that your scripts depend on. This statement performs two important tasks. 1) It communicates to other developers which modules the script uses, and 2) it identifies the dependent modules that AWS PowerShell tools need to package with the script, as part of the deployment. For more information about the #Requires statement in PowerShell, see About requires. For more information about PowerShell deployment packages, see Deploy PowerShell Lambda functions with .zip file archives.

When your PowerShell Lambda function uses the AWS PowerShell cmdlets, be sure to set a #Requires statement that references the AWSPowerShell.NetCore module, which supports PowerShell Core—and not the AWSPowerShell module, which only supports Windows PowerShell. Also, be sure to use version 3.3.270.0 or newer of AWSPowerShell.NetCore which optimizes the cmdlet import process. If you use an older version, you'll experience longer cold starts. For more information, see AWS Tools for PowerShell.
Returning data

Some Lambda invocations are meant to return data back to their caller. For example, if an invocation was in response to a web request coming from API Gateway, then our Lambda function needs to return back the response. For PowerShell Lambda, the last object that's added to the PowerShell pipeline is the return data from the Lambda invocation. If the object is a string, the data is returned as is. Otherwise the object is converted to JSON by using the `ConvertTo-Json` cmdlet.

For example, consider the following PowerShell statement, which adds `$PSVersionTable` to the PowerShell pipeline:

```
$PSVersionTable
```

After the PowerShell script is finished, the last object in the PowerShell pipeline is the return data for the Lambda function. `$PSVersionTable` is a PowerShell global variable that also provides information about the running environment.
AWS Lambda context object in PowerShell

When Lambda runs your function, it passes context information by making a $LambdaContext variable available to the handler. This variable provides methods and properties with information about the invocation, function, and execution environment.

Context properties

- **FunctionName** – The name of the Lambda function.
- **FunctionVersion** – The version of the function.
- **InvokedFunctionArn** – The Amazon Resource Name (ARN) that’s used to invoke the function. Indicates if the invoker specified a version number or alias.
- **MemoryLimitInMB** – The amount of memory that’s allocated for the function.
- **AwsRequestId** – The identifier of the invocation request.
- **LogGroupName** – The log group for the function.
- **LogStreamName** – The log stream for the function instance.
- **RemainingTime** – The number of milliseconds left before the execution times out.
- **Identity** – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
- **ClientContext** – (mobile apps) Client context that’s provided to Lambda by the client application.
- **Logger** – The logger object for the function.

The following PowerShell code snippet shows a simple handler function that prints some of the context information.

```powershell
#Requires -Modules @{ModuleName='AWS.PowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function name:' $LambdaContext.FunctionName
Write-Host 'Remaining milliseconds:' $LambdaContext.RemainingTime.TotalMilliseconds
Write-Host 'Log group name:' $LambdaContext.LogGroupName
Write-Host 'Log stream name:' $LambdaContext.LogStreamName
```
AWS Lambda function logging in PowerShell

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see [Using Amazon CloudWatch logs with AWS Lambda](#).

This page describes how to produce log output from your Lambda function's code, or access logs using the AWS Command Line Interface, the Lambda console, or the CloudWatch console.

**Sections**

- Creating a function that returns logs
- Using the Lambda console
- Using the CloudWatch console
- Using the AWS Command Line Interface (AWS CLI)
- Deleting logs

### Creating a function that returns logs

To output logs from your function code, you can use cmdlets on `Microsoft.PowerShell.Utility`, or any logging module that writes to `stdout` or `stderr`. The following example uses `Write-Host`.

**Example function/Handler.ps1 – Logging**

```powershell
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host `## Environment variables
Write-Host AWS_LAMBDA_FUNCTION_VERSION=$Env:AWS_LAMBDA_FUNCTION_VERSION
Write-Host AWS_LAMBDA_LOG_GROUP_NAME=$Env:AWS_LAMBDA_LOG_GROUP_NAME
Write-Host AWS_LAMBDA_LOG_STREAM_NAME=$Env:AWS_LAMBDA_LOG_STREAM_NAME
Write-Host AWS_EXECUTION_ENV=$Env:AWS_EXECUTION_ENV
Write-Host AWS_LAMBDA_FUNCTION_NAME=$Env:AWS_LAMBDA_FUNCTION_NAME
Write-Host PATH=$Env:PATH
Write-Host `## Event
Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 3)
```

**Example log format**

```
START RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Version: $LATEST
```

---

Logfile: 56639408-xmpl-435f-9041-ac47ae25ceed

---

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see [Using Amazon CloudWatch logs with AWS Lambda](#).

This page describes how to produce log output from your Lambda function's code, or access logs using the AWS Command Line Interface, the Lambda console, or the CloudWatch console.

**Sections**

- Creating a function that returns logs
- Using the Lambda console
- Using the CloudWatch console
- Using the AWS Command Line Interface (AWS CLI)
- Deleting logs

### Creating a function that returns logs

To output logs from your function code, you can use cmdlets on `Microsoft.PowerShell.Utility`, or any logging module that writes to `stdout` or `stderr`. The following example uses `Write-Host`.

**Example function/Handler.ps1 – Logging**

```powershell
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host `## Environment variables
Write-Host AWS_LAMBDA_FUNCTION_VERSION=$Env:AWS_LAMBDA_FUNCTION_VERSION
Write-Host AWS_LAMBDA_LOG_GROUP_NAME=$Env:AWS_LAMBDA_LOG_GROUP_NAME
Write-Host AWS_LAMBDA_LOG_STREAM_NAME=$Env:AWS_LAMBDA_LOG_STREAM_NAME
Write-Host AWS_EXECUTION_ENV=$Env:AWS_EXECUTION_ENV
Write-Host AWS_LAMBDA_FUNCTION_NAME=$Env:AWS_LAMBDA_FUNCTION_NAME
Write-Host PATH=$Env:PATH
Write-Host `## Event
Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 3)
```

**Example log format**

```
START RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Version: $LATEST
```

---

Logfile: 56639408-xmpl-435f-9041-ac47ae25ceed

---

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see [Using Amazon CloudWatch logs with AWS Lambda](#).

This page describes how to produce log output from your Lambda function's code, or access logs using the AWS Command Line Interface, the Lambda console, or the CloudWatch console.

**Sections**

- Creating a function that returns logs
- Using the Lambda console
- Using the CloudWatch console
- Using the AWS Command Line Interface (AWS CLI)
- Deleting logs

### Creating a function that returns logs

To output logs from your function code, you can use cmdlets on `Microsoft.PowerShell.Utility`, or any logging module that writes to `stdout` or `stderr`. The following example uses `Write-Host`.

**Example function/Handler.ps1 – Logging**

```powershell
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host `## Environment variables
Write-Host AWS_LAMBDA_FUNCTION_VERSION=$Env:AWS_LAMBDA_FUNCTION_VERSION
Write-Host AWS_LAMBDA_LOG_GROUP_NAME=$Env:AWS_LAMBDA_LOG_GROUP_NAME
Write-Host AWS_LAMBDA_LOG_STREAM_NAME=$Env:AWS_LAMBDA_LOG_STREAM_NAME
Write-Host AWS_EXECUTION_ENV=$Env:AWS_EXECUTION_ENV
Write-Host AWS_LAMBDA_FUNCTION_NAME=$Env:AWS_LAMBDA_FUNCTION_NAME
Write-Host PATH=$Env:PATH
Write-Host `## Event
Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 3)
```

**Example log format**

```
START RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Version: $LATEST
```
Creating a function that returns logs

The .NET runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details.

REPORT line data fields

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function.
• **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.

• **XRAY TraceId** – For traced requests, the [AWS X-Ray trace ID](https).

• **SegmentId** – For traced requests, the X-Ray segment ID.

• **Sampled** – For traced requests, the sampling result.

**Using the Lambda console**

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded Code editor, you will find logs in the execution results. When you use the console test feature to invoke a function, you'll find Log output in the Details section.

**Using the CloudWatch console**

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

**To view logs on the CloudWatch console**

1. Open the Log groups page on the CloudWatch console.
2. Choose the log group for your function (/aws/lambda/your-function-name).
3. Choose a log stream.

Each log stream corresponds to an instance of your function. A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see [Error processor sample application for AWS Lambda](https).

**Using the AWS Command Line Interface (AWS CLI)**

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:
You can use the AWS CLI to retrieve logs for an invocation using the --log-type command option. The response contains a LogResult field that contains up to 4 KB of base64-encoded logs from the invocation.

**Example retrieve a log ID**

The following example shows how to retrieve a log ID from the LogResult field for a function named my-function.

```bash
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{
  "StatusCode": 200,
  "LogResult":
  "U1RBU1QgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTEzTgtOGNkYS0yOTc0YzVlNGZlMjEgVmVyc2lvb...",
  "ExecutedVersion": "$LATEST"
}
```

**Example decode the logs**

In the same command prompt, use the base64 utility to decode the logs. The following example shows how to retrieve base64-encoded logs for my-function.

```bash
aws lambda invoke --function-name my-function out --log-type Tail \
  --query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --decode
```

The cli-binary-format option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the AWS Command Line Interface User Guide for Version 2.

You should see the following output:
The base64 utility is available on Linux, macOS, and Ubuntu on Windows. macOS users may need to use `base64 -D`.

**Example get-logs.sh script**

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-reference-core-commands.html) in the AWS Command Line Interface User Guide for Version 2.

```bash
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --payload '{"key": "value"}' out
sed -i'' -e 's////g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-name stream1 --limit 5
```

**Example macOS and Linux (only)**

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```bash
chmod -R 755 get-logs.sh
```
Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
  "StatusCode": 200,
  "ExecutedVersion": "$LATEST"
}
{
  "events": [
    {
      "timestamp": 1559763003171,
      "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST\n",
      "ingestionTime": 1559763003309
    },
    {
      "timestamp": 1559763003173,
      "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf\tINFO\tENVIRONMENT VARIABLES\r\n{\r  "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
  "...",
      "ingestionTime": 1559763018353
    },
    {
      "timestamp": 1559763003173,
      "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf\tINFO\t\r\n{\r  "key": "value"}
",
      "ingestionTime": 1559763018353
    },
    {
      "timestamp": 1559763003218,
      "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
      "ingestionTime": 1559763018353
    },
    {
      "timestamp": 1559763003218,
      "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75 MB\n",
      "ingestionTime": 1559763018353
    }
  ]
}
```
Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which logs are deleted automatically.
AWS Lambda function errors in PowerShell

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

This page describes how to view Lambda function invocation errors for the PowerShell runtime using the Lambda console and the AWS CLI.

Sections

- Syntax
- How it works
- Using the Lambda console
- Using the AWS Command Line Interface (AWS CLI)
- Error handling in other AWS services
- What's next?

Syntax

Consider the following PowerShell script example statement:

```powershell
throw 'The Account is not found'
```

When you invoke this Lambda function, it throws a terminating error, and AWS Lambda returns the following error message:

```json
{
   "errorMessage": "The Account is not found",
   "errorType": "RuntimeException"
}
```

Note the `errorType` is `RuntimeException`, which is the default exception thrown by PowerShell. You can use custom error types by throwing the error like this:

```powershell
throw @{'Exception'='AccountNotFound';'Message'='The Account is not found'}
```

The error message is serialized with `errorType` set to `AccountNotFound`:
If you don't need an error message, you can throw a string in the format of an error code. The error code format requires that the string starts with a character and only contain letters and digits afterwards, with no spaces or symbols.

For example, if your Lambda function contains the following:

```java
throw 'AccountNotFound'
```

The error is serialized like this:

```json
{
  "errorMessage": "AccountNotFound",
  "errorType": "AccountNotFound"
}
```

### How it works

When you invoke a Lambda function, Lambda receives the invocation request and validates the permissions in your execution role, verifies that the event document is a valid JSON document, and checks parameter values.

If the request passes validation, Lambda sends the request to a function instance. The Lambda runtime environment converts the event document into an object, and passes it to your function handler.

If Lambda encounters an error, it returns an exception type, message, and HTTP status code that indicates the cause of the error. The client or service that invoked the Lambda function can handle the error programmatically, or pass it along to an end user. The correct error handling behavior depends on the type of application, the audience, and the source of the error.

The following list describes the range of status codes you can receive from Lambda.
2xx

A 2xx series error with a X-Amz-Function-Error header in the response indicates a Lambda runtime or function error. A 2xx series status code indicates that Lambda accepted the request, but instead of an error code, Lambda indicates the error by including the X-Amz-Function-Error header in the response.

4xx

A 4xx series error indicates an error that the invoking client or service can fix by modifying the request, requesting permission, or by retrying the request. 4xx series errors other than 429 generally indicate an error with the request.

5xx

A 5xx series error indicates an issue with Lambda, or an issue with the function's configuration or resources. 5xx series errors can indicate a temporary condition that can be resolved without any action by the user. These issues can't be addressed by the invoking client or service, but a Lambda function's owner may be able to fix the issue.

For a complete list of invocation errors, see InvokeFunction errors.

Using the Lambda console

You can invoke your function on the Lambda console by configuring a test event and viewing the output. The output is captured in the function's execution logs and, when active tracing is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the Functions page of the Lambda console.
2. Choose the function to test, and choose Test.
3. Under Test event, select New event.
4. Select a Template.
5. For Name, enter a name for the test. In the text entry box, enter the JSON test event.
6. Choose Save changes.
7. Choose Test.
The Lambda console invokes your function synchronously and displays the result. To see the response, logs, and other information, expand the Details section.

### Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface (AWS CLI) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

When you invoke a Lambda function in the AWS CLI, the AWS CLI splits the response into two documents. The AWS CLI response is displayed in your command prompt. If an error has occurred, the response contains a FunctionError field. The invocation response or error returned by the function is written to an output file. For example, `output.json` or `output.txt`.

The following `invoke` command example demonstrates how to invoke a function and write the invocation response to an `output.txt` file.

```bash
aws lambda invoke  
  --function-name my-function  
  --cli-binary-format raw-in-base64-out  
    --payload '{"key1": "value1", "key2": "value2", "key3": "value3"}' output.txt
```

The cli-binary-format option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#) in the [AWS Command Line Interface User Guide for Version 2](#).

You should see the AWS CLI response in your command prompt:

```json
{
    "StatusCode": 200,
    "FunctionError": "Unhandled",
    "ExecutedVersion": "$LATEST"
}
```
You should see the function invocation response in the `output.txt` file. In the same command prompt, you can also view the output in your command prompt using:

```
cat output.txt
```

You should see the invocation response in your command prompt.

Lambda also records up to 256 KB of the error object in the function's logs. For more information, see [AWS Lambda function logging in PowerShell](https://docs.aws.amazon.com/lambda/latest/dg/powerhell-logging.html).

### Error handling in other AWS services

When another AWS service invokes your function, the service chooses the invocation type and retry behavior. AWS services can invoke your function on a schedule, in response to a lifecycle event on a resource, or to serve a request from a user. Some services invoke functions asynchronously and let Lambda handle errors, while others retry or pass errors back to the user.

For example, API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502 error code. To customize the error response, you must catch errors in your code and format a response in the required format.

We recommend using AWS X-Ray to determine the source of an error and its cause. X-Ray allows you to find out which component encountered an error, and see details about the errors. The following example shows a function error that resulted in a 502 response from API Gateway.

For more information, see [Using AWS Lambda with AWS X-Ray](https://docs.aws.amazon.com/lambda/latest/dg/lambda-xray.html).

### What's next?

- Learn how to show logging events for your Lambda function on the [section called “Logging”](https://docs.aws.amazon.com/lambda/latest/dg/lambda-xray.html) page.
Building Lambda functions with Rust

Because Rust compiles to native code, you don't need a dedicated runtime to run Rust code on Lambda. Instead, use the Rust runtime client to build your project locally, and then deploy it to Lambda using the provided.al2023 or provided.al2 runtime. When you use provided.al2023 or provided.al2, Lambda automatically keeps the operating system up to date with the latest patches.

Note

The Rust runtime client is an experimental package. It is subject to change and intended only for evaluation purposes.

Tools and libraries for Rust

- **AWS SDK for Rust**: The AWS SDK for Rust provides Rust APIs to interact with Amazon Web Services infrastructure services.
- **Rust runtime client for Lambda**: The Rust runtime client is an experimental package. It is subject to breaking changes and not recommended for production.
- **Cargo Lambda**: This library provides a command line application to work with Lambda functions built with Rust.
- **Lambda HTTP**: This library provides a wrapper to work with HTTP events.
- **Lambda Extension**: This library provides support to write Lambda Extensions with Rust.
- **AWS Lambda Events**: This library provides type definitions for common event source integrations.

Sample Lambda applications for Rust

- **Basic Lambda function**: A Rust function that shows how to process basic events.
- **Lambda function with error handling**: A Rust function that shows how to handle custom Rust errors in Lambda.
- **Lambda function with shared resources**: A Rust project that initializes shared resources before creating the Lambda function.
- **Lambda HTTP events**: A Rust function that handles HTTP events.
• **Lambda HTTP events with CORS headers**: A Rust function that uses Tower to inject CORS headers.

• **Lambda REST API**: A REST API that uses Axum and Diesel to connect to a PostgreSQL database.

• **Serverless Rust demo**: A Rust project that shows the use of Lambda's Rust libraries, logging, environment variables, and the AWS SDK.

• **Basic Lambda Extension**: A Rust extension that shows how to process basic extension events.

• **Lambda Logs Amazon Kinesis Data Firehose Extension**: A Rust extension that shows how to send Lambda logs to Kinesis Data Firehose.

**Topics**

• **Lambda function handler in Rust**

• **Lambda context object in Rust**

• **Processing HTTP events with Rust**

• **Deploy Rust Lambda functions with .zip file archives**

• **Lambda function logging in Rust**

• **Lambda function errors in Rust**
Lambda function handler in Rust

Note

The Rust runtime client is an experimental package. It is subject to change and intended only for evaluation purposes.

The Lambda function handler is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

Write your Lambda function code as a Rust executable. Implement the handler function code and a main function and include the following:

- The lambda_runtime crate from crates.io, which implements the Lambda programming model for Rust.
- Include Tokio in your dependencies. The Rust runtime client for Lambda uses Tokio to handle asynchronous calls.

Example — Rust handler that processes JSON events

The following example uses the serde_json crate to process basic JSON events:

```rust
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};

async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

Note the following:
• **use**: Imports the libraries that your Lambda function requires.

• **async fn main**: The entry point that runs the Lambda function code. The Rust runtime client uses [Tokio](https://tokio.rs) as an async runtime, so you must annotate the main function with `#[tokio::main].`

• **async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error>**: This is the Lambda handler signature. It includes the code that runs when the function is invoked.

• **LambdaEvent<Value>**: This is a generic type that describes the event received by the Lambda runtime as well as the [Lambda function context](https://docs.aws.amazon.com/lambda/latest/dg/lambda-conventions.html).

• **Result<Value, Error>**: The function returns a Result type. If the function is successful, the result is a JSON value. If the function is not successful, the result is an error.

### Using shared state

You can declare shared variables that are independent of your Lambda function's handler code. These variables can help you load state information during the **Init phase**, before your function receives any events.

**Example — Share Amazon S3 client across function instances**

Note the following:

• **use aws_sdk_s3::Client**: This example requires you to add `aws-sdk-s3 = "0.26.0"` to the list of dependencies in your Cargo.toml file.

• **aws_config::from_env**: This example requires you to add `aws-config = "0.55.1"` to the list of dependencies in your Cargo.toml file.

```rust
use aws_sdk_s3::Client;
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde::{Deserialize, Serialize};

#[derive(Deserialize)]
struct Request {
    bucket: String,
}

#[derive(Serialize)]
struct Response {
    keys: Vec<String>,
}
```
async fn handler(client: &Client, event: LambdaEvent<Request>) -> Result<Response, Error> {
    let bucket = event.payload.bucket;
    let objects = client.list_objects_v2().bucket(bucket).send().await?
    let keys = objects
        .contents()
        .map(|s| s.iter().flat_map(|o| o.key().map(String::from)).collect())
        .unwrap_or_default();
    Ok(Response { keys })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    let shared_config = aws_config::from_env().load().await;
    let client = Client::new(&shared_config);
    let shared_client = &client;
    lambda_runtime::run(service_fn(move |event: LambdaEvent<Request>| async move {
        handler(&shared_client, event).await
    })).await
}
Lambda context object in Rust

Note

The Rust runtime client is an experimental package. It is subject to change and intended only for evaluation purposes.

When Lambda runs your function, it adds a context object to the LambdaEvent that the handler receives. This object provides properties with information about the invocation, function, and execution environment.

Context properties

- request_id: The AWS request ID generated by the Lambda service.
- deadline: The execution deadline for the current invocation in milliseconds.
- invoked_function_arn: The Amazon Resource Name (ARN) of the Lambda function being invoked.
- xray_trace_id: The AWS X-Ray trace ID for the current invocation.
- client_content: The client context object sent by the AWS mobile SDK. This field is empty unless the function is invoked using an AWS mobile SDK.
- identity: The Amazon Cognito identity that invoked the function. This field is empty unless the invocation request to the Lambda APIs was made using AWS credentials issued by Amazon Cognito identity pools.
- env_config: The Lambda function configuration from the local environment variables. This property includes information such as the function name, memory allocation, version, and log streams.

Accessing invoke context information

Lambda functions have access to metadata about their environment and the invocation request. The LambdaEvent object that your function handler receives includes the context metadata:

```rust
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};
```
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    let invoked_function_arn = event.context.invoked_function_arn;
    Ok(json!({ "message": format!("Hello, this is function
        {invoked_function_arn}!") })
    )
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
Processing HTTP events with Rust

Note

The **Rust runtime client** is an experimental package. It is subject to change and intended only for evaluation purposes.

Amazon API Gateway APIs, Application Load Balancers, and **Lambda function URLs** can send HTTP events to Lambda. You can use the **aws_lambda_events** crate from crates.io to process events from these sources.

**Example — Handle API Gateway proxy request**

Note the following:

- **use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse}**: The **aws_lambda_events** crate includes many Lambda events. To reduce compilation time, use feature flags to activate the events you need. Example:
  
  ```
  aws_lambda_events = { version = "0.8.3", default-features = false, features = ["apigw"] }
  ```

- **use http::HeaderMap**: This import requires you to add the **http** crate to your dependencies.

```rust
use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse};
use http::HeaderMap;
use lambda_runtime::{service_fn, Error, LambdaEvent};

async fn handler(
    event: LambdaEvent<ApiGatewayProxyRequest>,
) -> Result<ApiGatewayProxyResponse, Error> {
    let mut headers = HeaderMap::new();
    headers.insert("content-type", "text/html".parse().unwrap());
    let resp = ApiGatewayProxyResponse {
        status_code: 200,
        multi_value_headers: headers.clone(),
        is_base64_encoded: Some(false),
        body: Some("Hello AWS Lambda HTTP request".into()),
        headers,
    };
}
```
Ok(resp)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}

The Rust runtime client for Lambda also provides an abstraction over these event types that allows you to work with native HTTP types, regardless of which service sends the events. The following code is equivalent to the previous example, and it works out of the box with Lambda function URLs, Application Load Balancers, and API Gateway.

**Note**
The `lambda_http` crate uses the `lambda_runtime` crate underneath. You don't have to import `lambda_runtime` separately.

**Example — Handle HTTP requests**

```rust
use lambda_http::{service_fn, Error, IntoResponse, Request, RequestExt, Response};

async fn handler(event: Request) -> Result<impl IntoResponse, Error> {
    let resp = Response::builder()
        .status(200)
        .header("content-type", "text/html")
        .body("Hello AWS Lambda HTTP request")
        .map_err(Box::new)?;
    Ok(resp)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_http::run(service_fn(handler)).await
}
```

For another example of how to use `lambda_http`, see the [http-axum code sample](https://github.com/aws-labs/lambda-http-example/tree/main/http-axum) on the AWS Labs GitHub repository.
Sample HTTP Lambda events for Rust

- **Lambda HTTP events**: A Rust function that handles HTTP events.
- **Lambda HTTP events with CORS headers**: A Rust function that uses Tower to inject CORS headers.
- **Lambda HTTP events with shared resources**: A Rust function that uses shared resources initialized before the function handler is created.
Deploy Rust Lambda functions with .zip file archives

Note

The Rust runtime client is an experimental package. It is subject to change and intended only for evaluation purposes.

This page describes how to compile your Rust function, and then deploy the compiled binary to AWS Lambda using Cargo Lambda. It also shows how to deploy the compiled binary with the AWS Command Line Interface and the AWS Serverless Application Model CLI.

Sections

- Prerequisites
- Building Rust functions on macOS, Windows, or Linux
- Deploying the Rust function binary with Cargo Lambda
- Invoking your Rust function with Cargo Lambda

Prerequisites

- Rust
- AWS Command Line Interface (AWS CLI) version 2

Building Rust functions on macOS, Windows, or Linux

The following steps demonstrate how to create the project for your first Lambda function with Rust and compile it with Cargo Lambda.

1. Install Cargo Lambda, a Cargo subcommand, that compiles Rust functions for Lambda on macOS, Windows, and Linux.

   To install Cargo Lambda on any system that has Python 3 installed, use pip:

   ```bash
   pip3 install cargo-lambda
   ```

   To install Cargo Lambda on macOS or Linux, use Homebrew:
To install Cargo Lambda on Windows, use **Scoop**:

```bash
scoop bucket add cargo-lambda
scoop install cargo-lambda/cargo-lambda
```

For other options, see [Installation](#) in the Cargo Lambda documentation.

2. Create the package structure. This command creates some basic function code in `src/main.rs`. You can use this code for testing or replace it with your own.

```bash
cargo lambda new my-function
```

3. Inside the package's root directory, run the **build** subcommand to compile the code in your function.

```bash
cargo lambda build --release
```

(Optional) If you want to use AWS Graviton2 on Lambda, add the `--arm64` flag to compile your code for ARM CPUs.

```bash
cargo lambda build --release --arm64
```

4. Before deploying your Rust function, configure AWS credentials on your machine.

```bash
aws configure
```

**Deploying the Rust function binary with Cargo Lambda**

Use the **deploy** subcommand to deploy the compiled binary to Lambda. This command creates an execution role and then creates the Lambda function. To specify an existing execution role, use the `--iam-role` flag.

```bash
cargo lambda deploy my-function
```
Deploying your Rust function binary with the AWS CLI

You can also deploy your binary with the AWS CLI.

1. Use the `build` subcommand to build the .zip deployment package.

   ```bash
cargo lambda build --release --output-format zip
   ```

2. Deploy the .zip package to Lambda. For `--role`, specify the ARN of the execution role.

   ```bash
aws lambda create-function --function-name my-function
   --runtime provided.al2023
   --role arn:aws:iam::111122223333:role/lambda-role
   --handler rust.handler
   --zip-file fileb://target/lambda/my-function/bootstrap.zip
   ```

Deploying your Rust function binary with the AWS SAM CLI

You can also deploy your binary with the AWS SAM CLI.

1. Create an AWS SAM template with the resource and property definition. For more information, see `AWS::Serverless::Function` in the AWS Serverless Application Model Developer Guide.

   **Example SAM resource and property definition for a Rust binary**

   ```yaml
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: SAM template for Rust binaries
Resources:
  RustFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: target/lambda/my-function/
      Handler: rust.handler
      Runtime: provided.al2023
    Outputs:
      RustFunction:
        Description: "Lambda Function ARN"
        Value: !GetAtt RustFunction.Arn
   ```

2. Use the `build` subcommand to compile the function.
cargo lambda build --release

3. Use the `sam deploy` command to deploy the function to Lambda.

    sam deploy --guided

For more information about building Rust functions with the AWS SAM CLI, see [Building Rust Lambda functions with Cargo Lambda](https://docs.aws.amazon.com/lambda/latest/dg/rust-cargo-lambda.html) in the *AWS Serverless Application Model Developer Guide*.

## Invoking your Rust function with Cargo Lambda

Use the `invoke` subcommand to test your function with a payload.

    cargo lambda invoke --remote --data-ascii '{"command": "Hello world"}' my-function

## Invoking your Rust function with the AWS CLI

You can also use the AWS CLI to invoke the function.

    aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --payload '{"command": "Hello world"}' /tmp/out.txt

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/index.html) in the *AWS Command Line Interface User Guide for Version 2*.
AWS Lambda

Developer Guide

Lambda function logging in Rust
Note
The Rust runtime client is an experimental package. It is subject to change and intended
only for evaluation purposes.

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon
CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream
for each instance of your function. The Lambda runtime environment sends details about each
invocation to the log stream, and relays logs and other output from your function's code. For more
information, see Using Amazon CloudWatch logs with AWS Lambda. This page describes how to
produce log output from your Lambda function's code.

Creating a function that writes logs
To output logs from your function code, you can use any logging function that writes to stdout or
stderr, such as the println! macro. The following example uses println! to print a message
when the function handler starts and before it ﬁnishes.
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
println!("Rust function invoked");
let payload = event.payload;
let first_name = payload["firstName"].as_str().unwrap_or("world");
println!("Rust function responds to {}", &first_name);
Ok(json!({ "message": format!("Hello, {first_name}!") }))
}
#[tokio::main]
async fn main() -> Result<(), Error> {
lambda_runtime::run(service_fn(handler)).await
}

Advanced logging with the Tracing crate
Tracing is a framework for instrumenting Rust programs to collect structured, event-based
diagnostic information. This framework provides utilities to customize logging output levels and
Logging

959


formats, like creating structured JSON log messages. To use this framework, you must initialize a subscriber before implementing the function handler. Then, you can use tracing macros like debug, info, and error, to specify the level of logging that you want for each scenario.

Example — Using the Tracing crate

Note the following:

- `tracing_subscriber::fmt().json()`: When this option is included, logs are formatted in JSON. To use this option, you must include the json feature in the `tracing-subscriber` dependency (for example, `tracing-subscriber = { version = "0.3.11", features = ["json"] }`).
- `#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]`: This annotation generates a span every time the handler is invoked. The span adds the request ID to each log line.
- `{ %first_name }`: This construct adds the `first_name` field to the log line where it's used. The value for this field corresponds to the variable with the same name.

```rust
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde_json::{json, Value};

#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    tracing::info!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    tracing::info!(
        { %first_name },
        "Rust function responds to event"
    );
    Ok(json!(
        { "message": format!("Hello, {first_name}!") } )
    )
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt().json()
        .with_max_level(tracing::Level::INFO)
        // this needs to be set to remove duplicated information in the log.
        .with_current_span(false)
        // this needs to be set to false, otherwise ANSI color codes will
        // show up in a confusing manner in CloudWatch logs.
        .with_ansi(false)
        // disabling time is handy because CloudWatch will add the ingestion time.
        .without_time()
```
// remove the name of the function from every log entry
.with_target(false)
.init();
lambda_runtime::run(service_fn(handler)).await
}

When this Rust function is invoked, it prints two log lines similar to the following:

```json
{"level": "INFO", "fields": {"message": "Rust function invoked"}, "spans": [{"req_id": "45daaa7-1a72-470c-9a62-e79860044bb5", "name": "handler"}]}  
{"level": "INFO", "fields": {"message": "Rust function responds to event", "first_name": "David"}, "spans": [{"req_id": "45daaa7-1a72-470c-9a62-e79860044bb5", "name": "handler"}]}
```
Lambda function errors in Rust

Note
The Rust runtime client is an experimental package. It is subject to change and intended only for evaluation purposes.

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output. The Rust runtime client also writes the error in the log. The error appears in Amazon CloudWatch Logs by default. This page demonstrates how to return errors in your Lambda function's output.

Creating a function that returns errors

The following code sample shows a Lambda function that returns an error. The Rust Runtime handles this error directly.

Example

```rust
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde_json::{json, Value};
async fn handler(_event: LambdaEvent<Value>) -> Result<Value, String> {
    Err("something went wrong!").into()
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

This code returns the following error payload:

```json
{
    "errorType": "alloc::string::String",
    "errorMessage": "something went wrong!"
}
```

For a more advanced error handling example, see the sample application in the AWS Labs GitHub repository.
Using AWS Lambda with other services

AWS Lambda integrates with other AWS services to invoke functions or take other actions. These are some common use cases:

- Invoke a function in response to resource lifecycle events, such as with Amazon Simple Storage Service (Amazon S3). For more information, see Using AWS Lambda with Amazon S3.
- Respond to incoming HTTP requests. For more information, see Tutorial: Using Lambda with API Gateway.
- Consume events from a queue. For more information, see Using Lambda with Amazon SQS.
- Run a function on a schedule. For more information, see Using AWS Lambda with Amazon EventBridge (CloudWatch Events).

Depending on which service you're using with Lambda, the invocation generally works in one of two ways. An event drives the invocation or Lambda polls a queue or data stream and invokes the function in response to activity in the queue or data stream. Lambda integrates with Amazon Elastic File System and AWS X-Ray in a way that doesn't involve invoking functions.

For more information, see Event-driven invocation and Lambda polling. Or, look up the service that you want to work with in the following section to find a link to information about using that service with Lambda.

You can also use Lambda functions to interact programmatically with other AWS services using one of the AWS Software Development Kits (SDKs). For example, you can have a Lambda function create an Amazon S3 bucket or write data to a DynamoDB table using an API call from within your function. To learn more about AWS SDKs, see Tools to build on AWS.

Listing of services and links to more information

Find the service that you want to work with in the following table, to determine which method of invocation you should use. Follow the link from the service name to find information about how to set up the integration between the services. These topics also include example events that you can use to test your function.
Entries in this table are alphabetical by service name, excluding the "Amazon" or "AWS" prefix. You can also use your browser's search functionality to find your service in the list.

<table>
<thead>
<tr>
<th>Service</th>
<th>Method of invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Alexa</td>
<td>Event-driven; synchronous invocation</td>
</tr>
<tr>
<td>Amazon Managed Streaming for Apache Kafka</td>
<td>Lambda polling</td>
</tr>
<tr>
<td>Self-managed Apache Kafka</td>
<td>Lambda polling</td>
</tr>
<tr>
<td>Amazon API Gateway</td>
<td>Event-driven; synchronous invocation</td>
</tr>
<tr>
<td>AWS CloudFormation</td>
<td>Event-driven; asynchronous invocation</td>
</tr>
<tr>
<td>Amazon CloudFront (Lambda@Edge)</td>
<td>Event-driven; synchronous invocation</td>
</tr>
<tr>
<td>Amazon EventBridge (CloudWatch Events)</td>
<td>Event-driven; asynchronous invocation</td>
</tr>
<tr>
<td>Amazon CloudWatch Logs</td>
<td>Event-driven; asynchronous invocation</td>
</tr>
<tr>
<td>AWS CodeCommit</td>
<td>Event-driven; asynchronous invocation</td>
</tr>
<tr>
<td>AWS CodePipeline</td>
<td>Event-driven; asynchronous invocation</td>
</tr>
<tr>
<td>Amazon Cognito</td>
<td>Event-driven; synchronous invocation</td>
</tr>
<tr>
<td>AWS Config</td>
<td>Event-driven; asynchronous invocation</td>
</tr>
<tr>
<td>Amazon Connect</td>
<td>Event-driven; synchronous invocation</td>
</tr>
<tr>
<td>Amazon DynamoDB</td>
<td>Lambda polling</td>
</tr>
<tr>
<td>Amazon Elastic File System</td>
<td>Special integration</td>
</tr>
<tr>
<td>Service</td>
<td>Method of invocation</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>Elastic Load Balancing (Application Load Balancer)</td>
<td>Event-driven; synchronous invocation</td>
</tr>
<tr>
<td>AWS IoT</td>
<td>Event-driven; asynchronous invocation</td>
</tr>
<tr>
<td>AWS IoT Events</td>
<td>Event-driven; asynchronous invocation</td>
</tr>
<tr>
<td>Amazon Kinesis</td>
<td>Lambda polling</td>
</tr>
<tr>
<td>Amazon Kinesis Data Firehose</td>
<td>Event-driven; synchronous invocation</td>
</tr>
<tr>
<td>Amazon Lex</td>
<td>Event-driven; synchronous invocation</td>
</tr>
<tr>
<td>Amazon MQ</td>
<td>Lambda polling</td>
</tr>
<tr>
<td>Amazon Simple Email Service</td>
<td>Event-driven; asynchronous invocation</td>
</tr>
<tr>
<td>Amazon Simple Notification Service</td>
<td>Event-driven; asynchronous invocation</td>
</tr>
<tr>
<td>Amazon Simple Queue Service</td>
<td>Lambda polling</td>
</tr>
<tr>
<td>Amazon Simple Storage Service (Amazon S3)</td>
<td>Event-driven; asynchronous invocation</td>
</tr>
<tr>
<td>Amazon Simple Storage Service Batch</td>
<td>Event-driven; synchronous invocation</td>
</tr>
<tr>
<td>Secrets Manager</td>
<td>Event-driven; synchronous invocation</td>
</tr>
<tr>
<td>Amazon VPC Lattice</td>
<td>Event-driven; synchronous invocation</td>
</tr>
<tr>
<td>AWS X-Ray</td>
<td>Special integration</td>
</tr>
</tbody>
</table>
Event-driven invocation

Some services generate events that can invoke your Lambda function. For more information about designing these types of architectures, see Event driven architectures in Serverless Land.

When you implement an event-driven architecture, you grant the event-generating service permission to invoke your function in the function's resource-based policy. Then you configure that service to generate events that invoke your function.

The events are data structured in JSON format. The JSON structure varies depending on the service that generates it and the event type, but they all contain the data that the function needs to process the event.

Lambda converts the event document into an object and passes it to your function handler. For compiled languages, Lambda provides definitions for event types in a library. For more information, see the topic about building functions with your language: Building Lambda functions with C#, Building Lambda functions with Go, Building Lambda functions with Java, or Building Lambda functions with PowerShell.

Depending on the service, the event-driven invocation can be synchronous or asynchronous.

- For synchronous invocation, the service that generates the event waits for the response from your function. That service defines the data that the function needs to return in the response. The service controls the error strategy, such as whether to retry on errors. For more information, see the section called “Synchronous invocation”.

- For asynchronous invocation, Lambda queues the event before passing it to your function. When Lambda queues the event, it immediately sends a success response to the service that generated the event. After the function processes the event, Lambda doesn’t return a response to the event-generating service. For more information, see the section called “Asynchronous invocation”.

For more information about how Lambda manages error handling for synchronously and asynchronously invoked functions, see the section called “Error handling”.

Lambda polling

For services that generate a queue or data stream, you set up an event source mapping in Lambda to have Lambda poll the queue or a data stream.
When you implement a Lambda polling architecture, you grant Lambda permission to access the other service in the function’s execution role. Lambda reads data from the other service, creates an event, and invokes your function.
Common Lambda application types and use cases

Lambda functions and triggers are the core components of building applications on AWS Lambda. A Lambda function is the code and runtime that process events, while a trigger is the AWS service or application that invokes the function. To illustrate, consider the following scenarios:

- **File processing** – Suppose you have a photo sharing application. People use your application to upload photos, and the application stores these user photos in an Amazon S3 bucket. Then, your application creates a thumbnail version of each user's photos and displays them on the user's profile page. In this scenario, you may choose to create a Lambda function that creates a thumbnail automatically. Amazon S3 is one of the supported AWS event sources that can publish object-created events and invoke your Lambda function. Your Lambda function code can read the photo object from the S3 bucket, create a thumbnail version, and then save it in another S3 bucket.

- **Data and analytics** – Suppose you are building an analytics application and storing raw data in a DynamoDB table. When you write, update, or delete items in a table, DynamoDB streams can publish item update events to a stream associated with the table. In this case, the event data provides the item key, event name (such as insert, update, and delete), and other relevant details. You can write a Lambda function to generate custom metrics by aggregating raw data.

- **Websites** – Suppose you are creating a website and you want to host the backend logic on Lambda. You can invoke your Lambda function over HTTP using Amazon API Gateway as the HTTP endpoint. Now, your web client can invoke the API, and then API Gateway can route the request to Lambda.

- **Mobile applications** – Suppose you have a custom mobile application that produces events. You can create a Lambda function to process events published by your custom application. For example, you can configure a Lambda function to process the clicks within your custom mobile application.

AWS Lambda supports many AWS services as event sources. For more information, see Using AWS Lambda with other services. When you configure these event sources to trigger a Lambda function, the Lambda function is invoked automatically when events occur. You define event source mapping, which is how you identify what events to track and which Lambda function to invoke.

The following are introductory examples of event sources and how the end-to-end experience works.
Example 1: Amazon S3 pushes events and invokes a Lambda function

Amazon S3 can publish events of different types, such as PUT, POST, COPY, and DELETE object events on a bucket. Using the bucket notification feature, you can configure an event source mapping that directs Amazon S3 to invoke a Lambda function when a specific type of event occurs.

The following is a typical sequence:

1. The user creates an object in a bucket.
2. Amazon S3 detects the object created event.
3. Amazon S3 invokes your Lambda function using the permissions provided by the execution role.
4. AWS Lambda runs the Lambda function, specifying the event as a parameter.

You configure Amazon S3 to invoke your function as a bucket notification action. To grant Amazon S3 permission to invoke the function, update the function's resource-based policy.

Example 2: AWS Lambda pulls events from a Kinesis stream and invokes a Lambda function

For poll-based event sources, AWS Lambda polls the source and then invokes the Lambda function when records are detected on that source.

- CreateEventSourceMapping
- UpdateEventSourceMapping

The following steps describe how a custom application writes records to a Kinesis stream:

1. The custom application writes records to a Kinesis stream.
2. AWS Lambda continuously polls the stream, and invokes the Lambda function when the service detects new records on the stream. AWS Lambda knows which stream to poll and which Lambda function to invoke based on the event source mapping you create in Lambda.
3. The Lambda function is invoked with the incoming event.

When working with stream-based event sources, you create event source mappings in AWS Lambda. Lambda reads items from the stream and invokes the function synchronously. You don't
need to grant Lambda permission to invoke the function, but it does need permission to read from the stream.

Example 2: AWS Lambda pulls events from a Kinesis stream and invokes a Lambda function
Using AWS Lambda with Alexa

You can use Lambda functions to build services that give new skills to Alexa, the Voice assistant on Amazon Echo. The Alexa Skills Kit provides the APIs, tools, and documentation to create these new skills, powered by your own services running as Lambda functions. Amazon Echo users can access these new skills by asking Alexa questions or making requests.

The Alexa Skills Kit is available on GitHub.

- [Alexa Skills Kit SDK for Java](#)
- [Alexa Skills Kit SDK for Node.js](#)
- [Alexa Skills Kit SDK for Python](#)

Example Alexa smart home event

```json
{
    "header": {
        "payloadVersion": "1",
        "namespace": "Control",
        "name": "SwitchOnOffRequest"
    },
    "payload": {
        "switchControlAction": "TURN_ON",
        "appliance": {
            "additionalApplianceDetails": {
                "key2": "value2",
                "key1": "value1"
            },
            "applianceId": "sampleId"
        },
        "accessToken": "sampleAccessToken"
    }
}
```

For more information, see [Host a custom skill as an AWS Lambda Function](#) in the *Build Skills with the Alexa Skills Kit* guide.
Using AWS Lambda with Amazon API Gateway

You can create a web API with an HTTP endpoint for your Lambda function by using Amazon API Gateway. API Gateway provides tools for creating and documenting web APIs that route HTTP requests to Lambda functions. You can secure access to your API with authentication and authorization controls. Your APIs can serve traffic over the internet or can be accessible only within your VPC.

Resources in your API define one or more methods, such as GET or POST. Methods have an integration that routes requests to a Lambda function or another integration type. You can define each resource and method individually, or use special resource and method types to match all requests that fit a pattern. A proxy resource catches all paths beneath a resource. The ANY method catches all HTTP methods.

This section explains general information on how to choose an API type, add an endpoint to your Lambda function, and information on events, permissions, responses, and error handling.

Sections
- Adding an endpoint to your Lambda function
- Proxy integration
- Event format
- Response format
- Permissions
- Handling errors with an API Gateway API
- Choosing an API type
- Sample applications
- Tutorial: Using Lambda with API Gateway
- AWS SAM template for an API Gateway application

Adding an endpoint to your Lambda function

To add a public endpoint to your Lambda function

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Under **Function overview**, choose **Add trigger**.

4. Select **API Gateway**.

5. Choose **Create an API** or **Use an existing API**.
   
   a. **New API**: For **API type**, choose **HTTP API**. For more information, see [API types](#).
   
   b. **Existing API**: Select the API from the dropdown menu or enter the API ID (for example, r3pmxmplak).

6. For **Security**, choose **Open**.

7. Choose **Add**.

---

**Proxy integration**

API Gateway APIs are comprised of stages, resources, methods, and integrations. The stage and resource determine the path of the endpoint:

### API path format

- `/prod/` – The `prod` stage and root resource.
- `/prod/user` – The `prod` stage and `user` resource.
- `/dev/{proxy+}` – Any route in the `dev` stage.
- `/` – (HTTP APIs) The default stage and root resource.

A Lambda integration maps a path and HTTP method combination to a Lambda function. You can configure API Gateway to pass the body of the HTTP request as-is (custom integration), or to encapsulate the request body in a document that includes all of the request information including headers, resource, path, and method.

### Event format

Amazon API Gateway invokes your function **synchronously** with an event that contains a JSON representation of the HTTP request. For a custom integration, the event is the body of the request. For a proxy integration, the event has a defined structure. The following example shows a proxy event from an API Gateway REST API.

**Example** [event.json](#) API Gateway proxy event (REST API)

```json
{

}```
"resource": "/",
"path": "/",
"httpMethod": "GET",
"requestContext": {
    "resourcePath": "/",
    "httpMethod": "GET",
    "path": "/Prod/",
    ...
},
"headers": {
    "accept": "text/html, application/xhtml+xml, application/xml;q=0.9, image/webp, image/apng,*/*;q=0.8, application/signed-exchange;v=b3;q=0.9",
    "accept-encoding": "gzip, deflate, br",
    "Host": "70ixmpl4fl.execute-api.us-east-2.amazonaws.com",
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.132 Safari/537.36",
    "X-Amzn-Trace-Id": "Root=1-5e66d96f-7491f09xmpl79d18acf3d050",
    ...
},
"multiValueHeaders": {
    "accept": [
        "text/html, application/xhtml+xml, application/xml;q=0.9, image/webp, image/apng,*/*;q=0.8, application/signed-exchange;v=b3;q=0.9"
    ],
    "accept-encoding": [
        "gzip, deflate, br"
    ],
    ...
},
"queryStringParameters": null,
"multiValueQueryStringParameters": null,
"pathParameters": null,
"stageVariables": null,
"body": null,
"isBase64Encoded": false
}

Response format

API Gateway waits for a response from your function and relays the result to the caller. For a custom integration, you define an integration response and a method response to convert the output from the function to an HTTP response. For a proxy integration, the function must respond with a representation of the response in a specific format.
The following example shows a response object from a Node.js function. The response object represents a successful HTTP response that contains a JSON document.

Example `index.mjs` – Proxy integration response object (Node.js)

```javascript
var response = {
    "statusCode": 200,
    "headers": {
        "Content-Type": "application/json"
    },
    "isBase64Encoded": false,
    "multiValueHeaders": {
        "X-Custom-Header": ["My value", "My other value"]
    },
    "body": "\n    \"TotalCodeSize\": 104330022,\n    \"FunctionCount\": 26\n"};
```

The Lambda runtime serializes the response object into JSON and sends it to the API. The API parses the response and uses it to create an HTTP response, which it then sends to the client that made the original request.

Example HTTP response

```html
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 55
Connection: keep-alive
x-amzn-RequestId: 32998fea-xmpl-4268-8c72-16138d629356
X-Custom-Header: My value
X-Custom-Header: My other value
X-Amzn-Trace-Id: Root=1-5e6aa925-ccecxmplbae116148e52f036
{
    "TotalCodeSize": 104330022,
    "FunctionCount": 26
}
```

Permissions

Amazon API Gateway gets permission to invoke your function from the function's resource-based policy. You can grant invoke permission to an entire API, or grant limited access to a stage, resource, or method.
When you add an API to your function by using the Lambda console, using the API Gateway console, or in an AWS SAM template, the function's resource-based policy is updated automatically. The following is an example function policy.

**Example function policy**

```
{
   "Version": "2012-10-17",
   "Id": "default",
   "Statement": [
      {
         "Sid": "nodejs-apig-functiongetEndpointPermissionProd-BWDBXMPLXE2F",
         "Effect": "Allow",
         "Principal": {
            "Service": "apigateway.amazonaws.com"
         },
         "Action": "lambda:InvokeFunction",
         "Condition": {
            "StringEquals": {
               "aws:SourceAccount": "111122223333"
            },
            "ArnLike": {
            }
         }
      }
   ]
}
```

You can manage function policy permissions manually with the following API operations:

- [AddPermission](#)
- [RemovePermission](#)
- [GetPolicy](#)

To grant invocation permission to an existing API, use the `add-permission` command.

```
aws lambda add-permission --function-name my-function \\
```
You should see the following output:

```
{
}
```

**Note**

If your function and API are in different regions, the region identifier in the source ARN must match the region of the function, not the region of the API. When API Gateway invokes a function, it uses a resource ARN that is based on the ARN of the API, but modified to match the function's region.

The source ARN in this example grants permission to an integration on the GET method of the root resource in the default stage of an API, with ID mnh1xmpli7. You can use an asterisk in the source ARN to grant permissions to multiple stages, methods, or resources.

**Resource patterns**

- mnh1xmpli7/*/*GET/* – GET method on all resources in all stages.
- mnh1xmpli7/prod/*ANY/user – ANY method on the user resource in the prod stage.
- mnh1xmpli7/*/*/* – Any method on all resources in all stages.

For details on viewing the policy and removing statements, see [Cleaning up resource-based policies](#).

**Handling errors with an API Gateway API**

API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an
error, or returns a response in the wrong format, API Gateway returns a 502. In both cases, the body of the response from API Gateway is \{"message": "Internal server error"\}.

**Note**

API Gateway does not retry any Lambda invocations. If Lambda returns an error, API Gateway returns an error response to the client.

The following example shows an X-Ray trace map for a request that resulted in a function error and a 502 from API Gateway. The client receives the generic error message.

To customize the error response, you must catch errors in your code and format a response in the required format.

**Example** *index.mjs* – Error formatting

```javascript
var formatError = function(error){
  var response = {
    "statusCode": error.statusCode,
    "headers": {
      "Content-Type": "text/plain",
      "x-amzn-ErrorType": error.code
    },
    "isBase64Encoded": false,
    "body": error.code + ": " + error.message
  }
  return response
}
```

API Gateway converts this response into an HTTP error with a custom status code and body. In the trace map, the function node is green because it handled the error.
Choosing an API type

API Gateway supports three types of APIs that invoke Lambda functions:

- **HTTP API** – A lightweight, low-latency RESTful API.
- **REST API** – A customizable, feature-rich RESTful API.
- **WebSocket API** – A web API that maintains persistent connections with clients for full-duplex communication.

HTTP APIs and REST APIs are both RESTful APIs that process HTTP requests and return responses. HTTP APIs are newer and are built with the API Gateway version 2 API. The following features are new for HTTP APIs:

**HTTP API features**

- **Automatic deployments** – When you modify routes or integrations, changes deploy automatically to stages that have automatic deployment enabled.
- **Default stage** – You can create a default stage (`$default`) to serve requests at the root path of your API's URL. For named stages, you must include the stage name at the beginning of the path.
- **CORS configuration** – You can configure your API to add CORS headers to outgoing responses, instead of adding them manually in your function code.

REST APIs are the classic RESTful APIs that API Gateway has supported since launch. REST APIs currently have more customization, integration, and management features.

**REST API features**

- **Integration types** – REST APIs support custom Lambda integrations. With a custom integration, you can send just the body of the request to the function, or apply a transform template to the request body before sending it to the function.
- **Access control** – REST APIs support more options for authentication and authorization.
• **Monitoring and tracing** – REST APIs support AWS X-Ray tracing and additional logging options.

For a detailed comparison, see [Choosing between HTTP APIs and REST APIs](#) in the *API Gateway Developer Guide*.

WebSocket APIs also use the API Gateway version 2 API and support a similar feature set. Use a WebSocket API for applications that benefit from a persistent connection between the client and API. WebSocket APIs provide full-duplex communication, which means that both the client and the API can send messages continuously without waiting for a response.

HTTP APIs support a simplified event format (version 2.0). The following example shows an event from an HTTP API.

**Example** [event-v2.json](#) – API Gateway proxy event (HTTP API)

```json
{
    "version": "2.0",
    "routeKey": "ANY /nodejs-apig-function-1G3XMPLZXVXYI",
    "rawPath": "/default/nodejs-apig-function-1G3XMPLZXVXYI",
    "rawQueryString": "",
    "cookies": [
        "s_fid=7AABXMPL1AFD9BBF-0643XMPL09956DE2",
        "regStatus=pre-register"
    ],
    "headers": {
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/;q=0.8,application/signed-exchange;v=b3;q=0.9",
        "accept-encoding": "gzip, deflate, br",
        ...
    },
    "requestContext": {
        "accountId": "123456789012",
        "apiId": "r3pmxmplak",
        "domainName": "r3pmxmplak.execute-api.us-east-2.amazonaws.com",
        "domainPrefix": "r3pmxmplak",
        "http": {
            "method": "GET",
            "path": "/default/nodejs-apig-function-1G3XMPLZXVXYI",
            "protocol": "HTTP/1.1",
            "sourceIp": "205.255.255.176",
            "userAgent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.132 Safari/537.36"
    }
}
```
For more information, see [AWS Lambda integrations](https://docs.aws.amazon.com/apigateway/latest/developerguide/lambda-integrations.html) in the API Gateway Developer Guide.

**Sample applications**

The GitHub repository for this guide provides the following sample application for API Gateway.

- **API Gateway with Node.js** – A function with an AWS SAM template that creates a REST API that has AWS X-Ray tracing enabled. It includes scripts for deploying, invoking the function, testing the API, and cleanup.

Lambda also provides [blueprints](https://github.com/aws-lambda-apigateway-blueprints) and [templates](https://github.com/aws-lambda-apigateway-templates) that you can use to create an API Gateway application in the Lambda console.

**Tutorial: Using Lambda with API Gateway**

In this tutorial, you create a REST API through which you invoke a Lambda function using an HTTP request. Your Lambda function will perform create, read, update, and delete (CRUD) operations on a DynamoDB table. This function is provided here for demonstration, but you will learn to configure an API Gateway REST API that can invoke any Lambda function.
Using API Gateway provides users with a secure HTTP endpoint to invoke your Lambda function and can help manage large volumes of calls to your function by throttling traffic and automatically validating and authorizing API calls. API Gateway also provides flexible security controls using AWS Identity and Access Management (IAM) and Amazon Cognito. This is useful for use cases where advance authorization is required for calls to your application.

To complete this tutorial, you will go through the following stages:

1. Create and configure a Lambda function in Python or Node.js to perform operations on a DynamoDB table.
2. Create a REST API in API Gateway to connect to your Lambda function.
3. Create a DynamoDB table and test it with your Lambda function in the console.
4. Deploy your API and test the full setup using curl in a terminal.

By completing these stages, you will learn how to use API Gateway to create an HTTP endpoint that can securely invoke a Lambda function at any scale. You will also learn how to deploy your API, and how to test it in the console and by sending an HTTP request using a terminal.
A sample AWS Serverless Application Model (AWS SAM) template for the Lambda application you create in this tutorial is also available. See [AWS SAM template for an API Gateway application](https://aws.amazon.com/sam).

**Sections**

- **Prerequisites**
- **Create a permissions policy**
- **Create an execution role**
- **Create the function**
- **Invoke the function using the AWS CLI**
- **Create a REST API using API Gateway**
- **Create a resource on your REST API**
- **Create an HTTP POST method**
- **Create a DynamoDB table**
- **Test the integration of API Gateway, Lambda, and DynamoDB**
- **Deploy the API**
- **Use curl to invoke your function using HTTP requests**
- **Clean up your resources (optional)**

**Prerequisites**

**Sign up for an AWS account**

If you do not have an AWS account, complete the following steps to create one.

**To sign up for an AWS account**

2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign...
administrative access to an administrative user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing My Account.

Create an administrative user

After you sign up for an AWS account, create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing Root user and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the AWS Sign-In User Guide.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the IAM User Guide.

Create an administrative user

- For your daily administrative tasks, grant administrative access to an administrative user in AWS IAM Identity Center.

  For instructions, see Getting started in the AWS IAM Identity Center User Guide.

Sign in as the administrative user

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the AWS Sign-In User Guide.
Install the AWS Command Line Interface

If you have not yet installed the AWS Command Line Interface, follow the steps at Installing or updating the latest version of the AWS CLI to install it.

The tutorial requires a command line terminal or shell to run commands. In Linux and macOS, use your preferred shell and package manager.

⚠️ Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux.

Create a permissions policy

Before you can create an execution role for your Lambda function, you first need to create a permissions policy to give your function permission to access the required AWS resources. For this tutorial, the policy allows Lambda to perform CRUD operations on a DynamoDB table and write to Amazon CloudWatch Logs.

To create the policy

1. Open the Policies page of the IAM console.
2. Choose Create Policy.
3. Choose the JSON tab, and then paste the following custom policy into the JSON editor.

```json
{

}```
"Version": "2012-10-17",
"Statement": [
  {
    "Sid": "Stmt1428341300017",
    "Action": [
      "dynamodb:DeleteItem",
      "dynamodb:GetItem",
      "dynamodb:PutItem",
      "dynamodb:Query",
      "dynamodb:Scan",
      "dynamodb:UpdateItem"
    ],
    "Effect": "Allow",
    "Resource": "*"
  },
  {
    "Sid": "",
    "Resource": "*",
    "Action": [
      "logs:CreateLogGroup",
      "logs:CreateLogStream",
      "logs:PutLogEvents"
    ],
    "Effect": "Allow"
  }
]

4. Choose **Next: Tags**.

5. Choose **Next: Review**.


7. Choose **Create policy**.
Create an execution role

An execution role is an AWS Identity and Access Management (IAM) role that grants a Lambda function permission to access AWS services and resources. To enable your function to perform operations on a DynamoDB table, you attach the permissions policy you created in the previous step.

To create an execution role and attach your custom permissions policy

1. Open the Roles page of the IAM console.
2. Choose Create role.
3. For the type of trusted entity, choose AWS service, then for the use case, choose Lambda.
4. Choose Next.
5. In the policy search box, enter lambda-apigateway-policy.
6. In the search results, select the policy that you created (lambda-apigateway-policy), and then choose Next.
7. Under Role details, for the Role name, enter lambda-apigateway-role, then choose Create role.

Later in the tutorial, you need the Amazon Resource Name (ARN) of the role you just created. On the Roles page of the IAM console, choose the name of your role (lambda-apigateway-role) and copy the Role ARN displayed on the Summary page.
Create the function

The following code example receives an event input from API Gateway specifying an operation to perform on the DynamoDB table you will create and some payload data. If the parameters the function receives are valid, it performs the requested operation on the table.

Node.js

Example index.mjs

```javascript
console.log('Loading function');

import { DynamoDBDocumentClient, PutCommand, GetCommand, UpdateCommand, DeleteCommand} from '@aws-sdk/lib-dynamodb';
import { DynamoDBClient } from '@aws-sdk/client-dynamodb';

const ddbClient = new DynamoDBClient({ region: 'us-west-2' });
const ddbDocClient = DynamoDBDocumentClient.from(ddbClient);

// Define the name of the DDB table to perform the CRUD operations on
const tablename = 'lambda-apigateway';

/**
 * Provide an event that contains the following keys:
 *
 * - operation: one of 'create,' 'read,' 'update,' 'delete,' or 'echo'
 * - payload: a JSON object containing the parameters for the table item
 * to perform the operation on
 */
export const handler = async (event, context) => {
    const operation = event.operation;
```
if (operation == 'echo'){
    return(event.payload);
}

else {
    event.payload.TableName = tablename;

    switch (operation) {
    case 'create':
        await ddbDocClient.send(new PutCommand(event.payload));
        break;
    case 'read':
        var table_item = await ddbDocClient.send(new
    GetCommand(event.payload));
        console.log(table_item);
        break;
    case 'update':
        await ddbDocClient.send(new UpdateCommand(event.payload));
        break;
    case 'delete':
        await ddbDocClient.send(new DeleteCommand(event.payload));
        break;
    default:
        return ('Unknown operation: ${operation}');
    }
}
}

### Note

In this example, the name of the DynamoDB table is defined as a variable in your function code. In a real application, best practice is to pass this parameter as an environment variable and to avoid hardcoding the table name. For more information see [Using AWS Lambda environment variables](#).

### To create the function

1. Save the code example as a file named `index.mjs` and, if necessary, edit the AWS region specified in the code. The region specified in the code must be the same as the region in which you create your DynamoDB table later in the tutorial.
2. Create a deployment package using the following `zip` command.

   ```
   zip function.zip index.mjs
   ```

3. Create a Lambda function using the `create-function` AWS CLI command. For the `role` parameter, enter the execution role's Amazon Resource Name (ARN) that you copied earlier.

   ```
   aws lambda create-function --function-name LambdaFunctionOverHttps
   --zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x
   --role arn:aws:iam::123456789012:role/service-role/lambda-apigateway-role
   ```

### Python 3

**Example LambdaFunctionOverHttps.py**

```python
import boto3
import json

# define the DynamoDB table that Lambda will connect to
tableName = "lambda-apigateway"

# create the DynamoDB resource
dynamo = boto3.resource('dynamodb').Table(tableName)

print('Loading function')

def handler(event, context):
    '''Provide an event that contains the following keys:

    - operation: one of the operations in the operations dict below
    - payload: a JSON object containing parameters to pass to the
      operation being performed
    
    # define the functions used to perform the CRUD operations
    def ddb_create(x):
        dynamo.put_item(**x)

    def ddb_read(x):
        dynamo.get_item(**x)

    def ddb_update(x):
    ```
dynamo.update_item(**x)

def ddb_delete(x):
    dynamo.delete_item(**x)

def echo(x):
    return x

operation = event['operation']

operations = {
    'create': ddb_create,
    'read': ddb_read,
    'update': ddb_update,
    'delete': ddb_delete,
    'echo': echo,
}

if operation in operations:
    return operations[operation](event.get('payload'))
else:
    raise ValueError('Unrecognized operation "{}"'.format(operation))

**Note**
In this example, the name of the DynamoDB table is defined as a variable in your function code. In a real application, best practice is to pass this parameter as an environment variable and to avoid hardcoding the table name. For more information see [Using AWS Lambda environment variables](#).

To create the function

1. Save the code example as a file named LambdaFunctionOverHttps.py.
2. Create a deployment package using the following `zip` command.

   ```bash
   zip function.zip LambdaFunctionOverHttps.py
   ```
3. Create a Lambda function using the `create-function` AWS CLI command. For the role parameter, enter the execution role's Amazon Resource Name (ARN) that you copied earlier.
Invoke the function using the AWS CLI

Before integrating your function with API Gateway, confirm that you have deployed the function successfully. Create a test event containing the parameters your API Gateway API will send to Lambda and use the AWS CLI invoke command to run your function.

To invoke the Lambda function with the AWS CLI

1. Save the following JSON as a file named `input.txt`.

   ```json
   {
       "operation": "echo",
       "payload": {
           "somekey1": "somevalue1",
           "somekey2": "somevalue2"
       }
   }
   ```

2. Run the following `invoke` AWS CLI command.

   ```bash
   aws lambda invoke --function-name LambdaFunctionOverHttps \
   --payload file://input.txt outputfile.txt --cli-binary-format raw-in-base64-out
   ```
The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-aws-index.html) in the *AWS Command Line Interface User Guide for Version 2*.

You should see the following response:

```json
{
    "StatusCode": 200,
    "ExecutedVersion": "LATEST"
}
```

3. Confirm that your function performed the echo operation you specified in the JSON test event. Inspect the `outputfile.txt` file and verify it contains the following:

```json
{"somekey1": "somevalue1", "somekey2": "somevalue2"}
```

---

**Create a REST API using API Gateway**

In this step, you create the API Gateway REST API you will use to invoke your Lambda function.

**To create the API**

1. Open the [API Gateway console](https://console.aws.amazon.com/apigateway).
2. Choose **Create API**.
3. In the **REST API** box, choose **Build**.
4. Under **API details**, leave **New API** selected, and for **API Name**, enter **DynamoDBOperations**.

---

---

---
5. Choose Create API.

**Create a resource on your REST API**

To add an HTTP method to your API, you first need to create a resource for that method to operate on. Here you create the resource to manage your DynamoDB table.

**To create the resource**

1. In the API Gateway console, on the Resources page for your API, choose Create Resource.
2. In Resource details, for Resource name enter DynamoDBManager.
3. Choose Create Resource.

**Create an HTTP POST method**

In this step, you create a method (POST) for your DynamoDBManager resource. You link this POST method to your Lambda function so that when the method receives an HTTP request, API Gateway invokes your Lambda function.
Note

For the purpose of this tutorial, one HTTP method (POST) is used to invoke a single Lambda function which carries out all of the operations on your DynamoDB table. In a real application, best practice is to use a different Lambda function and HTTP method for each operation. For more information, see The Lambda monolith in Serverless Land.

To create the POST method

1. On the Resources page for your API, ensure that the /DynamoDBManager resource is highlighted. Then, in the Methods pane, choose Create Method.
2. For Method type, choose POST.
3. For Integration type, leave Lambda function selected.
4. For Lambda function, choose the Amazon Resource Name (ARN) for your function (LambdaFunctionOverHttps).
5. Choose Create method.

Create a DynamoDB table

Create an empty DynamoDB table that your Lambda function will perform CRUD operations on.

To create the DynamoDB table

1. Open the Tables page of the DynamoDB console.
2. Choose Create table.
3. Under Table details, do the following:
1. For **Table name**, enter `lambda-apigateway`.

2. For **Partition key**, enter `id`, and keep the data type set as `String`.

4. Under **Table settings**, keep the **Default settings**.

5. Choose **Create table**.

### Test the integration of API Gateway, Lambda, and DynamoDB

You're now ready to test the integration of your API Gateway API method with your Lambda function and your DynamoDB table. Using the API Gateway console, you send requests directly to your POST method using the console's test function. In this step, you first use a create operation to add a new item to your DynamoDB table, then you use an update operation to modify the item.

**Test 1: To create a new item in your DynamoDB table**

1. In the **API Gateway console**, choose your API (DynamoDBOperations) and open the **Resources** page.

2. Choose the **Test** tab.

3. Under **Test method**, leave **Query strings** and **Headers** empty. For **Request body**, paste the following JSON:

   ```json
   {
     "operation": "create",
     "payload": {
       "Item": {
         "id": "1234ABCD",
         "number": 5
       }
     }
   }
   ```
4. Choose Test.

The results that are displayed when the test completes should show status 200. This status code indicates that the create operation was successful.

To confirm, check that your DynamoDB table now contains the new item.

5. Open the Tables page of the DynamoDB console and choose the lambda-apigateway table.

6. Choose Explore table items. In the Items returned pane, you should see one item with the id 1234ABCD and the number 5.

**Test 2: To update the item in your DynamoDB table**

1. In the API Gateway console, return to your POST method's Test tab.

2. Under Test method, leave Query strings and Headers empty. For Request body, paste the following JSON:

   ```json
   {
       "operation": "update",
       "payload": {
           "Key": {
               "id": "1234ABCD"
           },
           "AttributeUpdates": {
               "number": {
                   "Value": 10
               }
           }
       }
   }
   ```

3. Choose Test.

The results which are displayed when the test completes should show status 200. This status code indicates that the update operation was successful.

To confirm, check that the item in your DynamoDB table has been modified.

4. Open the Tables page of the DynamoDB console and choose the lambda-apigateway table.
5. Choose **Explore table items**. In the **Items returned** pane, you should see one item with the id 1234ABCD and the number 10.

**Deploy the API**

For a client to call the API, you must create a deployment and an associated stage. A stage represents a snapshot of your API including its methods and integrations.

**To deploy the API**

1. Open the **APIs** page of the [API Gateway console](https://aws.amazon.com) and choose the DynamoDBOperations API.
2. On the **Resources** page for your API choose **Deploy API**.
3. For **Stage**, choose *New stage*, then for **Stage name**, enter **test**.
4. Choose **Deploy**.
5. In the **Stage details** pane, copy the **Invoke URL**. You will use this in the next step to invoke your function using an HTTP request.
Use curl to invoke your function using HTTP requests

You can now invoke your Lambda function by issuing an HTTP request to your API. In this step, you will create a new item in your DynamoDB table and then delete it.

**To invoke the Lambda function using curl**

1. Run the following `curl` command using the invoke URL you copied in the previous step. When you use `curl` with the `-d (data)` option, it automatically uses the HTTP POST method.

   ```bash
   curl https://l8togsqxd8.execute-api.us-west-2.amazonaws.com/test/DynamoDBManager \
   -d '{"operation": "create", "payload": {"Item": {"id": "5678EFGH", "number": 15}}}'
   ```

2. To verify that the create operation was successful, do the following:
   1. Open the [Tables page](#) of the DynamoDB console and choose the `lambda-apigateway` table.
   2. Choose **Explore table items**. In the **Items returned** pane, you should see an item with the **id** `5678EFGH` and the **number** `15`.
   3. Run the following `curl` command to delete the item you just created. Use your own invoke URL.

      ```bash
      curl https://l8togsqxd8.execute-api.us-west-2.amazonaws.com/test/DynamoDBManager \
      -d '{"operation": "delete", "payload": {"Key": {"id": "5678EFGH"}}}'
      ```

3. Confirm that the delete operation was successful. In the **Items returned** pane of the DynamoDB console **Explore items** page, verify that the item with **id** `5678EFGH` is no longer in the table.
Clean up your resources (optional)

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you’re no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose Actions, Delete.
4. Type delete in the text input field and choose Delete.

To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose Delete.
4. Enter the name of the role in the text input field and choose Delete.

To delete the API

1. Open the APIs page of the API Gateway console.
2. Select the API you created.
3. Choose Actions, Delete.
4. Choose Delete.

To delete the DynamoDB table

1. Open the Tables page of the DynamoDB console.
2. Select the table you created.
3. Choose Delete.
4. Enter delete in the text box.
5. Choose Delete table.
AWS SAM template for an API Gateway application

Below is a sample AWS SAM template for the Lambda application from the tutorial. Copy the text below to a file and save it next to the ZIP package you created previously. Note that the Handler and Runtime parameter values should match the ones you used when you created the function in the previous section.

Example template.yaml

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  LambdaFunctionOverHttps:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs18.x
      Policies: AmazonDynamoDBFullAccess
    Events:
      HttpPost:
        Type: Api
        Properties:
          Path: '/DynamoDBOperations/DynamoDBManager'
          Method: post
```

For information on how to package and deploy your serverless application using the package and deploy commands, see Deploying serverless applications in the AWS Serverless Application Model Developer Guide.

Using AWS Lambda with AWS Application Composer

AWS Application Composer is a visual builder for designing modern applications on AWS. You design your application architecture by dragging, grouping, and connecting AWS services in a visual canvas. Application Composer creates infrastructure as code (IaC) templates from your design that you can deploy using AWS SAM or AWS CloudFormation.
Exporting a Lambda function to Application Composer

You can get started using Application Composer by creating a new project based on the configuration of an existing Lambda function using the Lambda console. To export your function's configuration and code to Application Composer to create a new project, do the following:

1. Open the Functions page of the Lambda console.
2. Select the function you want to use as a basis for your Application Composer project.
3. In the Function overview pane, choose Export to Application Composer.

   To export your function's configuration and code to Application Composer, Lambda creates an Amazon S3 bucket in your account to temporarily store this data.

4. In the dialog box, choose Confirm and create project to accept the default name for this bucket and export your function's configuration and code to Application Composer.

5. (Optional) To choose another name for the Amazon S3 bucket that Lambda creates, enter a new name and choose Confirm and create project. Amazon S3 bucket names must be globally unique and follow the bucket naming rules.

6. To save your project and function files in Application Composer, activate local sync mode.

   **Note**

   If you've used the Export to Application Composer feature before and created an Amazon S3 bucket using the default name, Lambda can re-use this bucket if it still exists. Accept the default bucket name in the dialog box to re-use the existing bucket.

Amazon S3 transfer bucket configuration

The Amazon S3 bucket that Lambda creates to transfer your function's configuration automatically encrypts objects using the AES 256 encryption standard. Lambda also configures the bucket to use the bucket owner condition to ensure that only your AWS account is able to add objects to the bucket.

Lambda configures the bucket to automatically delete objects 10 days after they are uploaded. However, Lambda doesn't automatically delete the bucket itself. To delete the bucket from your AWS account, follow the instructions in Deleting a bucket. The default bucket name uses the prefix lambdasam, a 10-digit alphanumeric string, and the AWS Region you created your function in:
To avoid additional charges being added to your AWS account, we recommend that you delete the Amazon S3 bucket as soon as you have finished exporting your function to Application Composer. Standard Amazon S3 pricing applies.

**Required permissions**

To use the Lambda integration with Application Composer feature, you need certain permissions to download an AWS SAM template and to write your function's configuration to Amazon S3.

To download an AWS SAM template, you must have permission to use the following API actions:

- `iam:GetPolicy`
- `iam:GetPolicyVersion`
- `iam:GetRole`
- `iam:GetRolePolicy`
- `iam:ListAttachedRolePolicies`
- `iam:ListRolePolicies`
- `iam:ListRoles`

You can grant permission to use all of these actions by adding the `AWSLambda_ReadOnlyAccess` AWS managed policy to your IAM user role.

For Lambda to write your function's configuration to Amazon S3, you must have permission to use the following API actions:

- `S3:PutObject`
- `S3:CreateBucket`
- `S3:PutBucketEncryption`
- `S3:PutBucketLifecycleConfiguration`

If you are unable to export your function's configuration to Application Composer, check that your account has the required permissions for these operations. If you have the required permissions,
but still cannot export your function's configuration, check for any resource-based policies that might limit access to Amazon S3.

**Other resources**

For a more detailed tutorial on how to design a serverless application in Application Composer based on an existing Lambda function, see the section called "Infrastructure as code (IaC)".

To use Application Composer and AWS SAM to design and deploy a complete serverless application using Lambda, you can also follow the AWS Application Composer tutorial in the AWS Serverless Patterns Workshop.
Using AWS Lambda with AWS CloudTrail

AWS CloudTrail is a service that provides a record of actions taken by a user, role, or an AWS service. CloudTrail captures API calls as events. For an ongoing record of events in your AWS account, you create a trail. A trail enables CloudTrail to deliver log files of events to an Amazon S3 bucket.

You can take advantage of Amazon S3’s bucket notification feature and direct Amazon S3 to publish object-created events to AWS Lambda. Whenever CloudTrail writes logs to your S3 bucket, Amazon S3 can then invoke your Lambda function by passing the Amazon S3 object-created event as a parameter. The S3 event provides information, including the bucket name and key name of the log object that CloudTrail created. Your Lambda function code can read the log object and process the access records logged by CloudTrail. For example, you might write Lambda function code to notify you if specific API call was made in your account.

In this scenario, CloudTrail writes access logs to your S3 bucket. As for AWS Lambda, Amazon S3 is the event source so Amazon S3 publishes events to AWS Lambda and invokes your Lambda function.

Example CloudTrail log

```
{
   "Records": [
   {
      "eventVersion": "1.02",
      "userIdentity": {
         "type": "Root",
         "principalId": "123456789012",
         "arn": "arn:aws:iam::123456789012:root",
         "accountId": "123456789012",
         "accessKeyId": "access-key-id",
         "sessionContext": {
            "attributes": {
               "mfaAuthenticated": "false",
               "creationDate": "2015-01-24T22:41:54Z"
            }
         }
      },
      "eventTime": "2015-01-24T23:26:50Z",
      "eventSource": "sns.amazonaws.com",
      "eventName": "CreateTopic",
```

"awsRegion": "us-east-2",
"sourceIPAddress": "205.251.233.176",
"userAgent": "console.amazonaws.com",
"requestParameters": {
    "name": "dropmeplease"
},
"responseElements": {
    "topicArn": "arn:aws:sns:us-east-2:123456789012:exampletopic"
},
"requestID": "3fdb7834-9079-557e-8ef2-350abc03536b",
"eventID": "17b46459-dada-427b-b8e2-5a4ca9ff1a9c",
"eventType": "AwsApiCall",
"recipientAccountId": "123456789012"
},
{
    "eventVersion": "1.02",
    "userIdentity": {
        "type": "Root",
        "principalId": "123456789012",
        "arn": "arn:aws:iam::123456789012:root",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2015-01-24T22:41:54Z"
            }
        }
    },
    "eventTime": "2015-01-24T23:27:02Z",
    "eventSource": "sns.amazonaws.com",
    "eventName": "GetTopicAttributes",
    "awsRegion": "us-east-2",
    "sourceIPAddress": "205.251.233.176",
    "userAgent": "console.amazonaws.com",
    "requestParameters": {
        "topicArn": "arn:aws:sns:us-east-2:123456789012:exampletopic"
    },
    "responseElements": null,
    "requestID": "4a0388f7-a0af-5df9-9587-c5c98c29cbe9",
    "eventID": "ec5bb073-8fa1-4d45-b03c-f07b9fc9ea18",
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"}
For detailed information about how to configure Amazon S3 as the event source, see Using AWS Lambda with Amazon S3.

Topics

- Logging Lambda API calls with CloudTrail
- Sample function code
Logging Lambda API calls with CloudTrail

Lambda is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Lambda. CloudTrail captures API calls for Lambda as events. The calls captured include calls from the Lambda console and code calls to the Lambda API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon Simple Storage Service (Amazon S3) bucket, including events for Lambda. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in Event history. Using the information collected by CloudTrail, you can determine the request that was made to Lambda, the IP address from which the request was made, who made the request, when it was made, and additional details.

For more information about CloudTrail, including how to configure and enable it, see the AWS CloudTrail User Guide.

Lambda information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When supported event activity occurs in Lambda, that activity is recorded in a CloudTrail event along with other AWS service events in Event history. You can view, search, and download recent events in your AWS account. For more information, see Viewing events with CloudTrail event history in the AWS CloudTrail User Guide.

For an ongoing record of events in your AWS account, including events for Lambda, you create a trail. A trail enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs.

For more information, see the following topics in the AWS CloudTrail User Guide:

- Overview for creating a trail
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple regions and Receiving CloudTrail log files from multiple accounts
Every log entry contains information about who generated the request. The user identity information in the log helps you determine whether the request was made with user credentials, with temporary security credentials for a role or federated user, or by another AWS service. For more information, see the `userIdentity` field in the CloudTrail event reference.

You can store your log files in your bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted by using Amazon S3 server-side encryption (SSE).

You can choose to have CloudTrail publish Amazon Simple Notification Service (Amazon SNS) notifications when new log files are delivered if you want to take quick action upon log file delivery. For more information, see Configuring Amazon SNS notifications for CloudTrail.

You can also aggregate Lambda log files from multiple Regions and multiple AWS accounts into a single S3 bucket. For more information, see Working with CloudTrail log files.

**List of supported Lambda API actions**

Lambda supports logging the following actions as events in CloudTrail log files.

- **AddLayerVersionPermission**
- **AddPermission**
- **CreateAlias**
- **CreateEventSourceMapping**
- **CreateFunction**
  
  (The Environment and ZipFile parameters are omitted from the CloudTrail logs for CreateFunction.)
- **CreateFunctionUrlConfig**

---

**Note**

In the CloudTrail log file, the `eventName` might include date and version information, but it is still referring to the same public API. For example, the `GetFunction` action might appear as "GetFunction20150331". To see the `eventName` for a particular action, view a log file entry in your event history. For more information, see Viewing events with CloudTrail event history in the AWS CloudTrail User Guide.
• **DeleteEventSourceMapping**
• **DeleteFunction**
• **DeleteFunctionUrlConfig**
• **GetEventSourceMapping**
• **GetFunction**
• **GetFunctionUrlConfig**
• **GetFunctionConfiguration**
• **GetLayerVersionPolicy**
• **GetPolicy**
• **ListEventSourceMappings**
• **ListFunctions**
• **ListFunctionUrlConfigs**
• **PublishLayerVersion**

(The ZipFile parameter is omitted from the CloudTrail logs for PublishLayerVersion.)

• **RemovePermission**
• **TagResource**
• **UntagResource**
• **UpdateAlias**
• **UpdateEventSourceMapping**
• **UpdateFunctionCode**

(The ZipFile parameter is omitted from the CloudTrail logs for UpdateFunctionCode.)

• **UpdateFunctionConfiguration**

(The Environment parameter is omitted from the CloudTrail logs for UpdateFunctionConfiguration.)

• **UpdateFunctionUrlConfig**

### Understanding Lambda log file entries

CloudTrail log files contain one or more log entries where each entry is made up of multiple JSON-formatted events. A log entry represents a single request from any source and includes information
about the requested action, any parameters, the date and time of the action, and so on. The log entries are not guaranteed to be in any particular order. That is, they are not an ordered stack trace of the public API calls.

The following example shows CloudTrail log entries for the GetFunction and DeleteFunction actions.

**Note**

The `eventName` might include date and version information, such as "GetFunction20150331", but it is still referring to the same public API.

```json
{
    "Records": [
        {
            "eventVersion": "1.03",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "A1B2C3D4E5F6G7EXAMPLE",
                "arn": "arn:aws:iam::999999999999:user/myUserName",
                "accountId": "999999999999",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "userName": "myUserName"
            },
            "eventTime": "2015-03-18T19:03:36Z",
            "eventSource": "lambda.amazonaws.com",
            "eventName": "GetFunction",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "Python-httplib2/0.8 (gzip)",
            "errorCode": "AccessDenied",
            "requestParameters": null,
            "responseElements": null,
            "requestID": "8aebcd0f-cd1-10e4-a4a2-e356da31e4ff",
            "eventID": "e92a3e85-8ecd-4d23-8074-843aabbe89bf",
            "eventType": "AwsApiCall",
            "recipientAccountId": "999999999999"
        }
    ]
}
```
Using CloudTrail to track function invocations

CloudTrail also logs data events. You can turn on data event logging so that you log an event every time Lambda functions are invoked. This helps you understand what identities are invoking the functions and the frequency of their invocations. For more information on this option, see Logging data events for trails.

Note

CloudTrail logs only authenticated and authorized requests. CloudTrail does not log requests that fail authentication (credentials are missing or the provided credentials are not valid) or requests with credentials that are not authorized to invoke the function.
## Using CloudTrail to troubleshoot disabled event sources

One data event that can be encountered is a LambdaESMDisabled event. There are five general categories of error that are associated with this event:

**RESOURCE_NOT_FOUND**

The resource specified in the request does not exist.

**FUNCTION_NOT_FOUND**

The function attached to the event source does not exist.

**REGION_NAME_NOT_VALID**

A Region name provided to the event source or function is invalid.

**AUTHORIZATION_ERROR**

Permissions have not been set, or are misconfigured.

**FUNCTION_IN_FAILED_STATE**

The function code does not compile, has encountered an unrecoverable exception, or a bad deployment has occurred.

These errors are included in the CloudTrail event message within the `serviceEventDetails` entity.

**Example serviceEventDetails entity**

```json
"serviceEventDetails":{
   "ESMDisableReason":"Lambda Function not found"
}
```

**Sample function code**

Sample code is available for the following languages.

**Topics**

- [Node.js](#)
Node.js

The following example processes CloudTrail logs, and sends a notification when an Amazon SNS topic was created.

Example index.js

```javascript
var aws = require('aws-sdk');
var zlib = require('zlib');
var async = require('async');

var EVENT_SOURCE_TO_TRACK = /sns.amazonaws.com/;
var EVENT_NAME_TO_TRACK = /CreateTopic/;
var DEFAULT_SNS_REGION = 'us-west-2';
var SNS_TOPIC_ARN = 'The ARN of your SNS topic';

var s3 = new aws.S3();
var sns = new aws.SNS({
    apiVersion: '2010-03-31',
    region: DEFAULT_SNS_REGION
});

exports.handler = function(event, context, callback) {
    var srcBucket = event.Records[0].s3.bucket.name;
    var srcKey = event.Records[0].s3.object.key;

    async.waterfall([
        function fetchLogFromS3(next){
            console.log('Fetching compressed log from S3...');
            s3.getObject({
                Bucket: srcBucket,
                Key: srcKey
            },
                next);
        },
        function uncompressLog(response, next){
            console.log("Uncompressing log...");
            zlib.gunzip(response.Body, next);
        },
        function publishNotifications(jsonBuffer, next) {
            console.log('Filtering log...');
            var json = jsonBuffer.toString();
            console.log('CloudTrail JSON from S3: ', json);
        }
    ],
    callback);
};
```
```javascript
var records;
try {
    records = JSON.parse(json);
} catch (err) {
    next('Unable to parse CloudTrail JSON: ' + err);
    return;
}
var matchingRecords = records
    .Records
    .filter(function(record) {
        return record.eventSource.match(EVENT_SOURCE_TO_TRACK)
            && record.eventName.match(EVENT_NAME_TO_TRACK);
    });

console.log('Publishing ' + matchingRecords.length + ' notification(s) in parallel...');
async.each(matchingRecords,
    function(record, publishComplete) {
        console.log('Publishing notification: ', record);
        sns.publish({
            Message:
                'Alert... SNS topic created: 
                TopicARN=' + record.responseElements.topicArn + '
                JSON.stringify(record),
            TopicArn: SNS_TOPIC_ARN
        }, publishComplete);
        },
    next
    );
}, function (err) {
    if (err) {
        console.error('Failed to publish notifications: ', err);
    } else {
        console.log('Successfully published all notifications.');
    } callback(null,"message");
});
```

Zip up the sample code to create a deployment package. For instructions, see Deploy Node.js Lambda functions with .zip file archives.
Using AWS Lambda with Amazon EventBridge (CloudWatch Events)

Note

Amazon EventBridge is the preferred way to manage your events. CloudWatch Events and EventBridge are the same underlying service and API, but EventBridge provides more features. Changes you make in either CloudWatch Events or EventBridge will appear in each console. For more information, see the Amazon EventBridge documentation.

EventBridge (CloudWatch Events) helps you to respond to state changes in your AWS resources. For more information about EventBridge, see What is Amazon EventBridge? in the Amazon EventBridge User Guide.

When your resources change state, they automatically send events into an event stream. With EventBridge (CloudWatch Events), you can create rules that match selected events in the stream and route them to your AWS Lambda function to take action. For example, you can automatically invoke an AWS Lambda function to log the state of an EC2 instance or AutoScaling group.

EventBridge (CloudWatch Events) invokes your function asynchronously with an event document that wraps the event from its source. The following example shows an event that originated from a database snapshot in Amazon Relational Database Service.

Example EventBridge (CloudWatch Events) event

```json
{
    "version": "0",
    "id": "fe8d3c65-xmpl-c5c3-2c87-81584709a377",
    "detail-type": "RDS DB Instance Event",
    "source": "aws.rds",
    "account": "123456789012",
    "time": "2020-04-28T07:20:20Z",
    "region": "us-east-2",
    "resources": [
        "arn:aws:rds:us-east-2:123456789012:db:rdz6xmpliljl1b1"
    ],
    "detail": {
        "EventCategories": [
            "backup"
        ]
    }
}
```
You can also create a Lambda function and direct AWS Lambda to invoke it on a regular schedule. You can specify a fixed rate (for example, invoke a Lambda function every hour or 15 minutes), or you can specify a Cron expression.

**Example EventBridge (CloudWatch Events) message event**

```
{
    "version": "0",
    "account": "123456789012",
    "region": "us-east-2",
    "detail": {},
    "detail-type": "Scheduled Event",
    "source": "aws.events",
    "time": "2019-03-01T01:23:45Z",
    "id": "cdc73f9d-aea9-11e3-9d5a-835b769c0d9c",
    "resources": [
        "arn:aws:events:us-east-2:123456789012:rule/my-schedule"
    ]
}
```

To configure EventBridge (CloudWatch Events) to invoke your function

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function
4. Set the trigger type to EventBridge (CloudWatch Events).
5. For Rule, choose Create a new rule.
6. Configure the remaining options and choose Add.

For more information on expressions schedules, see [Schedule expressions using rate or cron](#).
Each AWS account can have up to 100 unique event sources of the **EventBridge (CloudWatch Events)- Schedule** source type. Each of these can be the event source for up to five Lambda functions. That is, you can have up to 500 Lambda functions that can be executing on a schedule in your AWS account.

**Topics**

- [Tutorial: Using AWS Lambda with scheduled events](#)
- [Schedule expressions using rate or cron](#)

**Tutorial: Using AWS Lambda with scheduled events**

In this tutorial, you do the following:

- Create a Lambda function using the **Schedule a periodic check of any URL** blueprint. You configure the Lambda function to run every minute. Note that if the function returns an error, Lambda logs error metrics to Amazon CloudWatch.

- Configure a CloudWatch alarm on the **Errors** metric of your Lambda function to post a message to your Amazon SNS topic when AWS Lambda emits error metrics to CloudWatch. You subscribe to the Amazon SNS topics to get email notification. In this tutorial, you do the following to set this up:
  
  - Create an Amazon SNS topic.
  
  - Subscribe to the topic so you can get email notifications when a new message is posted to the topic.
  
  - In Amazon CloudWatch, set an alarm on the **Errors** metric of your Lambda function to publish a message to your SNS topic when errors occur.

**Prerequisites**

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console](#) to create your first Lambda function.

**Create a Lambda function**

1. Sign in to the AWS Management Console and open the AWS Lambda console at [https://console.aws.amazon.com/lambda/](https://console.aws.amazon.com/lambda/).
2. Choose **Create function**.

3. Choose **Use a blueprint**.

4. Open the **Select blueprint** dropdown list and choose the **Schedule a periodic check of any URL** blueprint.

5. Configure the following settings.
   
   - **Function name** – `lambda-canary`.
   - **Execution role** – Create a new role from AWS policy templates.
   - **Role name** – `lambda-eventbridge-role`.
   - **Policy templates** – Simple microservice permissions.
   - **Rule** – Create a new rule.
   - **Rule name** – `CheckWebsiteScheduledEvent`.
   - **Rule description** – `CheckWebsiteScheduledEvent trigger`.
   - **Rule type** – Schedule expression.
   - **Schedule expression** – `rate(1 minute)`.
   - **Environment variables**
     - **site** – https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
     - **expected** – What is AWS Lambda?

6. Choose **Create function**.

EventBridge (CloudWatch Events) emits an event every minute, based on the schedule expression. The event triggers the Lambda function, which verifies that the expected string appears in the specified page. For more information on expressions schedules, see [Schedule expressions using rate or cron](#).

**Test the Lambda function**

Test the function with a sample event provided by the Lambda console.

1. Open the **Functions page** of the Lambda console.

2. Choose the **lambda-canary** function.

3. Choose **Test**.

4. Create a new event using the **CloudWatch event template (cloudwatch-scheduled-event)**.
5. Choose Create event.
6. Choose Invoke.

The output from the function execution is shown at the top of the page.

**Create an Amazon SNS topic and subscribe to it**

Create an Amazon Simple Notification Service (Amazon SNS) topic to receive notifications when the canary function returns an error.

**To create a topic**

1. Open the Amazon SNS console.
2. Switch to the AWS Region where you created the Lambda function.
3. Choose Topics, and then choose Create topic.
4. Create a topic with the following settings.
   - Type – Standard.
   - Name – lambda-canary-notifications.
   - Display name – Canary.
5. Choose Create topic.
6. On the lambda-canary-notifications topic page, choose Create subscription.
7. Create a subscription with the following settings.
   - Protocol – Email.
   - Endpoint – Your email address.
8. Choose Create subscription.

Amazon SNS sends an email from Canary <no-reply@sns.amazonaws.com>, reflecting the friendly name of the topic. Use the link in the email to confirm your address.

**Configure an alarm**

Configure an alarm in Amazon CloudWatch that monitors the Lambda function and sends a notification when it fails.
To create an alarm

1. Open the CloudWatch console.
2. Switch to the AWS Region where you created the Lambda function.
3. Choose All alarms.
4. Choose Create alarm.
5. On the Specify metric and conditions page, choose Select metric.
6. In the Metrics search box, enter lambda-canary Errors.
7. Choose Lambda > By Function 1 Name.
8. Select the lambda-canary Errors metric.
10. Set the threshold to Greater/Equal than 1.
11. On the Configure actions page, add a notification with the following settings:
   - Alarm state trigger – In alarm
   - Send notification to... – lambda-canary-notifications
12. On the Add name and description page, enter the following:
   - Name – lambda-canary-alarm
   - Description – Lambda canary alarm
13. Choose Create alarm.

Test the alarm

Update the function configuration to cause the function to return an error, which triggers the alarm.

To trigger an alarm

1. Open the Functions page of the Lambda console.
2. Choose the lambda-canary function.
4. Set expected to 404.
5. Choose Save.
Wait a minute, and then check your email for a message from Amazon SNS.

**Clean up your resources**

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

**To delete the Lambda function**

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

**To delete the CloudWatch alarm**

1. Open the [All alarms page](#) of the CloudWatch console.
2. Select the alarm you created.
3. Choose **Actions, Delete**.
4. Choose **Delete**.

**To delete the Amazon SNS subscription**

1. Open the [Subscriptions page](#) of the Amazon SNS console.
2. Select the subscription you created.
3. Choose **Delete, Delete**.

**To delete the Amazon SNS topic**

1. Open the [Topics page](#) of the Amazon SNS console.
2. Select the topic you created.
3. Choose **Delete**.
4. Enter **delete me** in the text input field.
5. Choose **Delete**.
Schedule expressions using rate or cron

AWS Lambda supports standard rate and cron expressions for frequencies of up to once per minute. Rate expressions are simpler to define but do not offer the fine-grained schedule control that cron triggers support.

EventBridge (CloudWatch Events) rate expressions have the following format.

\[
\text{rate}(\text{Value Unit})
\]

Where \text{Value} is a positive integer and \text{Unit} can be minute(s), hour(s), or day(s). A rate expression starts when you create the schedule. For a singular value the unit must be singular (for example, rate(1 day)), otherwise plural (for example, rate(5 days)).

**Rate expression examples**

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Every 5 minutes</td>
<td>rate(5 minutes)</td>
</tr>
<tr>
<td>Every hour</td>
<td>rate(1 hour)</td>
</tr>
<tr>
<td>Every seven days</td>
<td>rate(7 days)</td>
</tr>
</tbody>
</table>

Cron expressions have the following format.

\[
\text{cron}(\text{Minutes Hours Day-of-month Month Day-of-week Year})
\]

**Cron expression examples**

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:15 AM (UTC+0) every day</td>
<td>cron(15 10 * * ? *)</td>
</tr>
<tr>
<td>6:00 PM (UTC+0) Monday through Friday</td>
<td>cron(0 18 * * MON-FRI *)</td>
</tr>
<tr>
<td>8:00 AM (UTC+0) on the first day of the month</td>
<td>cron(0 8 1 * ? *)</td>
</tr>
<tr>
<td>Frequency</td>
<td>Expression</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Every 10 min on weekdays</td>
<td><code>cron(0/10 * ? * MON-FRI *)</code></td>
</tr>
<tr>
<td>Every 5 minutes between 8:00 AM (UTC+0) and 5:55 PM (UTC+0) weekdays</td>
<td><code>cron(0/5 8-17 ? * MON-FRI *)</code></td>
</tr>
<tr>
<td>9:00 AM (UTC+0) on the first Monday of each month</td>
<td><code>cron(0 9 ? * 2#1 *)</code></td>
</tr>
</tbody>
</table>

Note the following:

- All scheduled events use UTC+0 time zone.
- If you are using the Lambda console, do not include the `cron` prefix in your expression.
- One of the day-of-month or day-of-week values must be a question mark (?).

For more information, see [Using Lambda with Amazon EventBridge Scheduler](#).
Using Lambda with CloudWatch Logs

You can use a Lambda function to monitor and analyze logs from an Amazon CloudWatch Logs log stream. Create subscriptions for one or more log streams to invoke a function when logs are created or match an optional pattern. Use the function to send a notification or persist the log to a database or storage.

CloudWatch Logs invokes your function asynchronously with an event that contains log data. The value of the data field is a Base64-encoded .gzip file archive.

Example CloudWatch Logs message event

```
{
   "awslogs": {
       "data":
       "ewogICAgIm1lc3NhZ2VUeXBlIjogIkRBVEFfTUTFU0FHRIsCiAgICAib3duZXIiOiAiMTIzNDU2Nzg5MDEyIiwKICAgICJsb2dHcm91cCI6I...
   }
}
```

When decoded and decompressed, the log data is a JSON document with the following structure:

Example CloudWatch Logs message data (decoded)

```
{
    "messageType": "DATA_MESSAGE",
    "owner": "123456789012",
    "logGroup": "/aws/lambda/echo-nodejs",
    "logStream": "2019/03/13/[$LATEST]94fa867e5374431291a7fc14e2f56ae7",
    "subscriptionFilters": [
        "LambdaStream_cloudwatchlogs-node"
    ],
    "logEvents": [
        {
            "id": "34622316099697884706540976068822859012661220141643892546",
            "timestamp": 1552518348220,
            "message": "REPORT RequestId: 6234bffe-149a-b642-81ff-2e8e376d8aff
```
\tDuration: 46.84 ms \tBilled Duration: 47 ms \tMemory Size: 192 MB \tMax Memory Used: 72 MB\n"
        }
    ]
}
```
For a sample application that uses CloudWatch Logs as a trigger, see Error processor sample application for AWS Lambda.
Using AWS Lambda with AWS CloudFormation

In an AWS CloudFormation template, you can specify a Lambda function as the target of a custom resource. Use custom resources to process parameters, retrieve configuration values, or call other AWS services during stack lifecycle events.

The following example invokes a function that's defined elsewhere in the template.

Example – Custom resource definition

```yaml
Resources:
  primerinvoke:
    Type: AWS::CloudFormation::CustomResource
    Version: "1.0"
    Properties:
      ServiceToken: !GetAtt primer.Arn
      FunctionName: !Ref randomerror
```

The service token is the Amazon Resource Name (ARN) of the function that AWS CloudFormation invokes when you create, update, or delete the stack. You can also include additional properties like FunctionName, which AWS CloudFormation passes to your function as is.

AWS CloudFormation invokes your Lambda function **asynchronously** with an event that includes a callback URL.

Example – AWS CloudFormation message event

```
{
  "RequestType": "Create",
  "ResponseURL": "https://cloudformation-custom-resource-response-useast1.s3-us-east-1.amazonaws.com/arn%3Aaws%3Acloudformation%3Aus-east-1%3A123456789012%3Astack/lambda-error-processor/1134083a-2608-1e91-9897-022501a2c456%7Cprimerinvoke%7C5d478078-13e9-baf0-464a-7ef285ecc786?AWSAccessKeyId=AKIAIL3FODN7EXAMPLE&Expires=1555451971&Signature=28UijZePE5I4dvukKQqM%2F9RF104%3D",
  "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-processor/1134083a-2608-1e91-9897-022501a2c456",
  "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
  "LogicalResourceId": "primerinvoke",
  "ResourceType": "AWS::CloudFormation::CustomResource",
```
The function is responsible for returning a response to the callback URL that indicates success or failure. For the full response syntax, see [Custom resource response objects](#).

**Example – AWS CloudFormation custom resource response**

```json
{
  "Status": "SUCCESS",
  "PhysicalResourceId": "2019/04/18/[LATEST]b3d1bf65f19ec610654e4d9b9de47a0",
  "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-processor/1134083a-2608-1e91-9897-022501a2c456",
  "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
  "LogicalResourceId": "primerinvoke"
}
```

AWS CloudFormation provides a library called `cfn-response` that handles sending the response. If you define your function within a template, you can require the library by name. AWS CloudFormation then adds the library to the deployment package that it creates for the function.

If your function that a Custom Resource uses has an [Elastic Network Interface](#) attached to it, add the following resources to the VPC policy where `region` is the Region the function is in without the dashes. For example, `us-east-1` is `useast1`. This will allow the Custom Resource to respond to the callback URL that sends a signal back to the AWS CloudFormation stack.

```text
arn:aws:s3:::cloudformation-custom-resource-response-region",
"arn:aws:s3:::cloudformation-custom-resource-response-region/*",
```

The following example function invokes a second function. If the call succeeds, the function sends a success response to AWS CloudFormation, and the stack update continues. The template uses the [AWS::Serverless::Function](#) resource type provided by AWS Serverless Application Model.

**Example** [error-processor/template.yml](#) – Custom resource function

```yaml
Transform: 'AWS::Serverless-2016-10-31'
```
Resources:

primer:
  Type: AWS::Serverless::Function

Properties:
  Handler: index.handler
  Runtime: nodejs16.x
  InlineCode: |
    var aws = require('aws-sdk');
    var response = require('cfn-response');
    exports.handler = function(event, context) {
      // For Delete requests, immediately send a SUCCESS response.
      if (event.RequestType == "Delete") {
        response.send(event, context, "SUCCESS");
        return;
      }
      var responseStatus = "FAILED";
      var responseData = {};
      var functionName = event.ResourceProperties.FunctionName
      var lambda = new aws.Lambda();
      lambda.invoke({ FunctionName: functionName }, function(err, invokeResult) {
        if (err) {
          responseData = {Error: "Invoke call failed"};
          console.log(responseData.Error + "\n", err);
        }
        else responseStatus = "SUCCESS";
        response.send(event, context, responseStatus, responseData);
      });
    }
  Description: Invoke a function to create a log stream.
  MemorySize: 128
  Timeout: 8
  Role: !GetAtt role.Arn
  Tracing: Active

If the function that the custom resource invokes isn't defined in a template, you can get the source code for cfn-response from [cfn-response module](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-service-resources.html) in the AWS CloudFormation User Guide.

For a sample application that uses a custom resource to ensure that a function's log group is created before another resource that depends on it, see [Error processor sample application for AWS Lambda](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/arn-cookbook-error-processing.html).

For more information about custom resources, see [Custom resources](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-resource-custom.html) in the AWS CloudFormation User Guide.
Using AWS Lambda with CloudFront Lambda@Edge

Lambda@Edge is an extension of AWS Lambda that lets you deploy Python and Node.js functions at Amazon CloudFront edge locations. A common use case of Lambda@Edge is to use functions to customize the content that your CloudFront distribution delivers to your end users. Invoking these functions closer to the viewer instead of on origin servers significantly reduces latency and improves the user experience.

When you associate a CloudFront distribution with a Lambda@Edge function, CloudFront intercepts requests and responses at CloudFront edge locations. CloudFront then invokes your Lambda function by sending an event. You can have CloudFront invoke your Lambda function when the following events occur:

- When CloudFront receives a request from a viewer (viewer request)
- Before CloudFront forwards a request to the origin (origin request)
- When CloudFront receives a response from the origin (origin response)
- Before CloudFront returns the response to the viewer (viewer response)

Note
Lambda@Edge supports a limited set of runtimes and features. For details, see Requirements and restrictions on Lambda functions in the Amazon CloudFront developer guide.

The following is an example of a CloudFront event.
Example CloudFront message event

```json
{
    "Records": [
        {
            "cf": {
                "config": {
                    "distributionId": "EDFDVBD6EXAMPLE"
                },
                "request": {
                    "clientIp": "2001:0db8:85a3:0:0:8a2e:0370:7334",
                    "method": "GET",
                    "uri": "/picture.jpg",
                    "headers": {
                        "host": [
                            {
                                "key": "Host",
                                "value": "d111111abcdef8.cloudfront.net"
                            }
                        ],
                        "user-agent": [
                            {
                                "key": "User-Agent",
                                "value": "curl/7.51.0"
                            }
                        ]
                    }
                }
            }
        }
    ]
}
```

For more information about using Lambda@Edge, see [Using CloudFront with Lambda@Edge](#).
Using AWS Lambda with AWS CodeCommit

You can create a trigger for an AWS CodeCommit repository so that events in the repository will invoke a Lambda function. For example, you can invoke a Lambda function when a branch or tag is created or when a push is made to an existing branch.

Example AWS CodeCommit message event

```json
{
   "Records": [
      {
         "awsRegion": "us-east-2",
         "codecommit": {
            "references": [
               {
                  "commit": "5e493c6f3067653f3d04eca608b4901eb227078",
                  "ref": "refs/heads/master"
               }
            ]
         },
         "eventId": "31ade2c7-f889-47c5-a937-1cf99e2790e9",
         "eventName": "ReferenceChanges",
         "eventPartNumber": 1,
         "eventSource": "aws:codecommit",
         "eventTime": "2019-03-12T20:58:25.400+0000",
         "eventTotalParts": 1,
         "eventTriggerConfigId": "0d17d6a4-efeb-46f3-b3ab-a63741badeb8",
         "eventTriggerName": "index.handler",
         "eventVersion": "1.0",
         "userIdentityARN": "arn:aws:iam::123456789012:user/intern"
      }
   ]
}
```

For more information, see Manage triggers for an AWS CodeCommit repository.
Using AWS Lambda with AWS CodePipeline

AWS CodePipeline is a service that enables you to create continuous delivery pipelines for applications that run on AWS. You can create a pipeline to deploy your Lambda application. You can also configure a pipeline to invoke a Lambda function to perform a task when the pipeline runs. When you create a Lambda application in the Lambda console, Lambda creates a pipeline that includes source, build, and deploy stages.

CodePipeline invokes your function asynchronously with an event that contains details about the job. The following example shows an event from a pipeline that invoked a function named my-function.

**Example CodePipeline event**

```json
{
   "CodePipeline.job": {
      "id": "c0d76431-b0e7-xmpl-97e3-e8ee786eb6f6",
      "accountId": "123456789012",
      "data": {
         "actionConfiguration": {
            "configuration": {
               "FunctionName": "my-function",
               "UserParameters": "{"KEY":"VALUE"}"
            }
         },
         "inputArtifacts": [
            {
               "name": "my-pipeline-SourceArtifact",
               "revision": "e0c7xmpl2308ca3071aa7bab414de234ab52eea",
               "location": {
                  "type": "S3",
                  "s3Location": {
                     "bucketName": "us-west-2-123456789012-my-pipeline",
                     "objectKey": "my-pipeline/test-api-2/Td0SFRV"
                  }
               }
            }
         ],
         "outputArtifacts": [
            {
               "name": "invokeOutput",
               "revision": null,
            }
         ]
      }
   }
}
```
To complete the job, the function must call the CodePipeline API to signal success or failure. The following example Node.js function uses the PutJobSuccessResult operation to signal success. It gets the job ID for the API call from the event object.

**Example index.js**

```javascript
var AWS = require('aws-sdk')
var codepipeline = new AWS.CodePipeline()

exports.handler = async (event) => {
  console.log(JSON.stringify(event, null, 2))
  var jobId = event['CodePipeline.job'].id
  var params = {
    jobId: jobId
  }
  return codepipeline.putJobSuccessResult(params).promise()
}
```

For asynchronous invocation, Lambda queues the message and retries if your function returns an error. Configure your function with a destination to retain events that your function could not process.
For a tutorial on how to configure a pipeline to invoke a Lambda function, see Invoke an AWS Lambda function in a pipeline in the AWS CodePipeline User Guide.

You can use AWS CodePipeline to create a continuous delivery pipeline for your Lambda application. CodePipeline combines source control, build, and deployment resources to create a pipeline that runs whenever you make a change to your application's source code.

For an alternate method of creating a pipeline with AWS Serverless Application Model and AWS CloudFormation, watch Automate your serverless application deployments on the Amazon Web Services YouTube channel.

Permissions

To invoke a function, a CodePipeline pipeline needs permission to use the following API operations:

- ListFunctions
- InvokeFunction

The default pipeline service role includes these permissions.

To complete a job, the function needs the following permissions in its execution role.

- codepipeline:PutJobSuccessResult
- codepipeline:PutJobFailureResult

These permissions are included in the AWSCodePipelineCustomActionAccess managed policy.
Working with Amazon CodeWhisperer in the Lambda console

Amazon CodeWhisperer is a general purpose, machine learning-powered code generator that provides you with code recommendations in real time. When activated in the Lambda console, CodeWhisperer automatically generates suggestions based on your existing code and comments. Your personalized recommendations can vary in size and scope, ranging from a single one-liner to fully formed functions.

For more information, see Setting up Amazon CodeWhisperer with AWS Lambda in the Amazon CodeWhisperer User Guide.
Using AWS Lambda with Amazon Cognito

The Amazon Cognito Events feature enables you to run Lambda functions in response to events in Amazon Cognito. Amazon Cognito provides authentication, authorization, and user management for your web and mobile apps. You can invoke a Lambda function in response to important events in Amazon Cognito. For example, using the Sync Trigger events, you can invoke a Lambda function that is published each time a dataset is synchronized. To learn more and walk through an example, see [Introducing Amazon Cognito Events: Sync Triggers](https://aws.amazon.com/blogs/mobile/introducing-amazon-cognito-events-sync-triggers/) in the Mobile Development blog.

Example Amazon Cognito message event

```json
{
   "datasetName": "datasetName",
   "eventType": "SyncTrigger",
   "region": "us-east-1",
   "identityId": "identityId",
   "datasetRecords": {
      "SampleKey2": {
         "newValue": "newValue2",
         "oldValue": "oldValue2",
         "op": "replace"
      },
      "SampleKey1": {
         "newValue": "newValue1",
         "oldValue": "oldValue1",
         "op": "replace"
      }
   },
   "identityPoolId": "identityPoolId",
   "version": 2
}
```

You configure event source mapping using Amazon Cognito event subscription configuration. For information about event source mapping and a sample event, see [Amazon Cognito events](https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-events.html) in the [Amazon Cognito Developer Guide](https://docs.aws.amazon.com/cognito/latest/developerguide/).
Using Lambda with Amazon Connect

You can use a Lambda function to process requests from Amazon Connect. You can use Amazon Connect to create a cloud contact center.

Amazon Connect invokes your Lambda function synchronously with an event that contains the request body and metadata.

Example Amazon Connect request event

```
{
  "Details": {
    "ContactData": {
      "Attributes": {},
      "Channel": "VOICE",
      "ContactId": "4a573372-1f28-4e26-b97b-XXXXXXXXXXX",
      "CustomerEndpoint": {
        "Address": "+1234567890",
        "Type": "TELEPHONE_NUMBER"
      },
      "InitialContactId": "4a573372-1f28-4e26-b97b-XXXXXXXXXXX",
      "InitiationMethod": "INBOUND | OUTBOUND | TRANSFER | CALLBACK",
      "InstanceARN": "arn:aws:connect:aws-region:1234567890:instance/c8c0e68d-2200-4265-82c0-XXXXXXXXXXX",
      "PreviousContactId": "4a573372-1f28-4e26-b97b-XXXXXXXXXXX",
      "Queue": {
        "Name": "PasswordReset"
      },
      "SystemEndpoint": {
        "Address": "+1234567890",
        "Type": "TELEPHONE_NUMBER"
      }
    },
    "Parameters": {
      "sentAttributeKey": "sentAttributeValue"
    }
  },
  "Name": "ContactFlowEvent"
}
```
For information about how to use Amazon Connect with Lambda, see Invoke Lambda functions in the Amazon Connect administrator guide.
Using Lambda with Amazon DocumentDB

You can use a Lambda function to process events in an Amazon DocumentDB (with MongoDB compatibility) change stream by configuring an Amazon DocumentDB cluster as an event source. Then, you can automate event-driven workloads by invoking your Lambda function each time that data changes with your Amazon DocumentDB cluster.

Note

Lambda supports version 4.0 and 5.0 of Amazon DocumentDB only. Lambda doesn’t support version 3.6. Also, for event source mappings, Lambda supports instance-based clusters and regional clusters only. Lambda doesn’t support elastic clusters or global clusters. This limitation doesn’t apply when using Lambda as a client to connect to Amazon DocumentDB. Lambda can connect to all cluster types to perform CRUD operations.

Lambda processes events from Amazon DocumentDB change streams sequentially in the order in which they arrive. Because of this, your function can handle only one concurrent invocation from DocumentDB at a time. To monitor your function, you can track its concurrency metrics.

Topics

- Example Amazon DocumentDB event
- Prerequisites and permissions
- Network configuration
- Creating an Amazon DocumentDB event source mapping (console)
- Creating an Amazon DocumentDB event source mapping (SDK or CLI)
- Polling and stream starting positions
- Monitoring your Amazon DocumentDB event source
- Tutorial: Using AWS Lambda with Amazon DocumentDB Streams

Example Amazon DocumentDB event

```json
{
}
```
"eventSourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:canaryclusterb2a659a2-qo5tcmqkc103",
"events": [
    {
        "event": {
            "_id": {
                "_data": "0163eeb6e700000009010000090000041e1"
            },
            "clusterTime": {
                "$timestamp": {
                    "t": 1676588775,
                    "i": 9
                }
            },
            "documentKey": {
                "_id": {
                    "$oid": "63eeb6e7d418cd98afb1c1d7"
                }
            },
            "fullDocument": {
                "_id": {
                    "$oid": "63eeb6e7d418cd98afb1c1d7"
                },
                "anyField": "sampleValue"
            },
            "ns": {
                "db": "test_database",
                "coll": "test_collection"
            },
            "operationType": "insert"
        }
    }
],
"eventSource": "aws:docdb"
}

For more information about the events in this example and their shapes, see [Change Events](#) on the MongoDB Documentation website.

**Prerequisites and permissions**

Before you can use Amazon DocumentDB as an event source for your Lambda function, note the following prerequisites. You must:
• **Have an existing Amazon DocumentDB cluster in the same AWS account and AWS Region as your function.** If you don't have an existing cluster, you can create one by following the steps in [Get Started with Amazon DocumentDB](https://docs.aws.amazon.com/AmazonDocumentDB/latest/ug/getting-started.html) in the *Amazon DocumentDB Developer Guide*. Alternatively, the first set of steps in [Tutorial: Using AWS Lambda with Amazon DocumentDB Streams](https://docs.aws.amazon.com/AmazonDocumentDB/latest/ug/tutorials-lambda-streams.html) guide you through creating a DocumentDB cluster with all the necessary prerequisites.

• **Allow Lambda to access the Amazon Virtual Private Cloud (Amazon VPC) resources associated with your Amazon DocumentDB cluster.** For more information, see [Network configuration](https).

• **Enable TLS on your Amazon DocumentDB cluster.** This is the default setting. If you disable TLS, then Lambda cannot communicate with your cluster.

• **Activate change streams on your Amazon DocumentDB cluster.** For more information, see [Using Change Streams with Amazon DocumentDB](https://docs.aws.amazon.com/AmazonDocumentDB/latest/ug/using-change-streams.html) in the *Amazon DocumentDB Developer Guide*.

• **Provide Lambda with credentials to access your Amazon DocumentDB cluster.** When setting up the event source, provide the [AWS Secrets Manager](https://docs.aws.amazon.com/secretsmanager/latest/userguide/what-is-secrets-manager.html) key that contains the authentication details (username and password) required to access your cluster. To provide this key during setup, do either of the following:
  - If you're using the Lambda console for setup, then provide the key in the **Secrets manager key** field.
  - If you're using the AWS Command Line Interface (AWS CLI) for setup, then provide this key in the `source-access-configurations` option. You can include this option with either the `create-event-source-mapping` command or the `update-event-source-mapping` command. For example:

    ```bash
    aws lambda create-event-source-mapping
    ...--source-access-configurations
    ...
    ```

• **Grant Lambda permissions to manage resources related to your Amazon DocumentDB stream.** Manually add the following permissions to your function's **execution role**:
  - `rds:DescribeDBClusters`
  - `rds:DescribeDBClusterParameters`
  - `rds:DescribeDBSubnetGroups`
  - `ec2:CreateNetworkInterface`
  - `ec2:DescribeNetworkInterfaces`
Keep the size of Amazon DocumentDB change stream events that you send to Lambda under 6 MB. Lambda supports payload sizes of up to 6 MB. If your change stream tries to send Lambda an event larger than 6 MB, then Lambda drops the message and emits the OversizedRecordCount metric. Lambda emits all metrics on a best-effort basis.

Note

While Lambda functions typically have a maximum timeout limit of 15 minutes, event source mappings for Amazon MSK, self-managed Apache Kafka, Amazon DocumentDB, and Amazon MQ for ActiveMQ and RabbitMQ only support functions with maximum timeout limits of 14 minutes. This constraint ensures that the event source mapping can properly handle function errors and retries.

Network configuration

Lambda must have access to the Amazon VPC resources associated with your Amazon DocumentDB cluster. If you configured a secret in Secrets Manager to authenticate Lambda with the clusters, then also deploy a VPC endpoint for Secrets Manager.

Alternatively, ensure that the VPC associated with your Amazon DocumentDB cluster includes one NAT gateway per public subnet. For more information, see Internet and service access for VPC-connected functions.

Configure your Amazon VPC security groups with the following rules (at minimum):

- Inbound rules
  - For the security groups that you specify for your event source, allow all traffic on the Amazon DocumentDB cluster port (27017).
• (VPC endpoints only) For the security groups associated with your VPC endpoints, allow all traffic on port 443 from the security groups that you specify for your event source.

• Outbound rules
  • For all destinations, allow all traffic on port 443.
  • For the security groups that you specify for your event source, allow all traffic on the Amazon DocumentDB cluster port (27017).

Creating an Amazon DocumentDB event source mapping (console)

For a Lambda function to read from an Amazon DocumentDB cluster's change stream, create an event source mapping. This section describes how to do this from the Lambda console. For AWS SDK and AWS CLI instructions, see the section called “Creating an Amazon DocumentDB event source mapping (SDK or CLI)“.

To create an Amazon DocumentDB event source mapping (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
4. Under Trigger configuration, in the dropdown list, choose DocumentDB.
5. Configure the required options, and then choose Add.

Lambda supports the following options for Amazon DocumentDB event sources:

• **DocumentDB cluster** – Select an Amazon DocumentDB cluster.

• **Activate trigger** – Choose whether you want to activate the trigger immediately. If you select this check box, then your function immediately starts receiving traffic from the specified Amazon DocumentDB change stream upon creation of the event source mapping. We recommend that you clear the check box to create the event source mapping in a deactivated state for testing. After creation, you can activate the event source mapping at any time.

• **Database name** – Enter the name of a database within the cluster to consume.

• (Optional) **Collection name** – Enter the name of a collection within the database to consume. If you don't specify a collection, then Lambda listens to all events from each collection in the database.
• **Batch size** – Set the maximum number of messages to retrieve in a single batch, up to 10,000. The default batch size is 100.

• **Starting position** – Choose the position in the stream to start reading records from.
  - **Latest** – Process only new records that are added to the stream. Your function starts processing records only after Lambda finishes creating your event source. This means that some records may be dropped until your event source is created successfully.
  - **Trim horizon** – Process all records in the stream. Lambda uses the log retention duration of your cluster to determine where to start reading events from. Specifically, Lambda starts reading from $current\_time - log\_retention\_duration$. Your change stream must already be active before this timestamp for Lambda to read all events properly.
  - **At timestamp** – Process records starting from a specific time. Your change stream must already be active before the specified timestamp for Lambda to read all events properly.

• **Authentication** – Choose the authentication method for accessing the brokers in your cluster.
  - **BASIC\_AUTH** – With basic authentication, you must provide the Secrets Manager key that contains the credentials to access your cluster.

• **Secrets Manager key** – Choose the Secrets Manager key that contains the authentication details (username and password) required to access your Amazon DocumentDB cluster.

• (Optional) **Batch window** – Set the maximum amount of time in seconds to gather records before invoking your function, up to 300.

• (Optional) **Full document configuration** – For document update operations, choose what you want to send to the stream. The default value is `Default`, which means that for each change stream event, Amazon DocumentDB sends only a delta describing the changes made. For more information about this field, see [FullDocument](#) in the MongoDB Javadoc API documentation.
  - **Default** – Lambda sends only a partial document describing the changes made.
  - **UpdateLookup** – Lambda sends a delta describing the changes, along with a copy of the entire document.

**Creating an Amazon DocumentDB event source mapping (SDK or CLI)**

To create or manage an Amazon DocumentDB event source mapping with an [AWS SDK](#), you can use the following API operations:

- [CreateEventSourceMapping](#)
- [ListEventSourceMappings](#)
• **GetEventSourceMapping**

• **UpdateEventSourceMapping**

• **DeleteEventSourceMapping**

---

**Note**

When you update, disable, or delete an event source mapping for Amazon DocumentDB, it can take up to 15 minutes for your changes to take effect. Before this period has elapsed, your event source mapping may continue to process events and invoke your function using your previous settings. This is true even when the status of the event source mapping displayed in the console indicates that your changes have been applied.

To create the event source mapping with the AWS CLI, use the `create-event-source-mapping` command. The following example uses this command to map a function named `my-function` to an Amazon DocumentDB change stream. The event source is specified by an Amazon Resource Name (ARN), with a batch size of 500, starting from the timestamp in Unix time. The command also specifies the Secrets Manager key that Lambda uses to connect to Amazon DocumentDB. Additionally, it includes `document-db-event-source-config` parameters that specify the database and the collection to read from.

```bash
aws lambda create-event-source-mapping --function-name my-function \
  --event-source-arn arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-epzcyyu4pjoy \
  --batch-size 500 \
  --starting-position AT_TIMESTAMP \
  --starting-position-timestamp 1541139109 \
  --source-access-configurations
    '[["Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-east-1:123456789012:secret:DocDBSecret-BAtjxi"]]' \
  --document-db-event-source-config '{"DatabaseName":"test_database", "CollectionName": "test_collection"}' \
```

You should see output that looks like this:

```
{
  "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
  "BatchSize": 500,
}```
After creation, you can use the `update-event-source-mapping` command to update the settings for your Amazon DocumentDB event source. The following example updates the batch size to 1,000 and the batch window to 10 seconds. For this command, you need the UUID of your event source mapping, which you can retrieve using the `list-event-source-mapping` command or the Lambda console.

```
aws lambda update-event-source-mapping --function-name my-function \
  --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
  --batch-size 1000 \
  --batch-window 10
```

You should see this output that looks like this:

```
{
  "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
  "BatchSize": 500,
  "DocumentDBEventSourceConfig": {
    "CollectionName": "test_collection",
    "DatabaseName": "test_database",
    "FullDocument": "Default"
  },
  "MaximumBatchingWindowInSeconds": 0,
  "LastModified": 1541359182.919,
  "LastProcessingResult": "OK",
}
```
Lambda updates settings asynchronously, so you may not see these changes in the output until the process completes. To view the current settings of your event source mapping, use the `get-event-source-mapping` command.

```bash
aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b
```

You should see this output that looks like this:

```json
{
  "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
  "DocumentDBEventSourceConfig": {
    "CollectionName": "test_collection",
    "DatabaseName": "test_database",
    "FullDocument": "Default"
  },
  "BatchSize": 1000,
  "MaximumBatchingWindowInSeconds": 10,
  "LastModified": 1541359182.919,
  "LastProcessingResult": "OK",
  "State": "Enabled",
  "StateTransitionReason": "User action"
}
```

To delete your Amazon DocumentDB event source mapping, use the `delete-event-source-mapping` command.

```bash
aws lambda delete-event-source-mapping
--uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284
```

### Polling and stream starting positions

Be aware that stream polling during event source mapping creation and updates is eventually consistent.
During event source mapping creation, it may take several minutes to start polling events from the stream.

During event source mapping updates, it may take several minutes to stop and restart polling events from the stream.

This behavior means that if you specify LATEST as the starting position for the stream, the event source mapping could miss events during creation or updates. To ensure that no events are missed, specify the stream starting position as TRIM_HORIZON or AT_TIMESTAMP.

**Monitoring your Amazon DocumentDB event source**

To help you monitor your Amazon DocumentDB event source, Lambda emits the `IteratorAge` metric when your function finishes processing a batch of records. *Iterator age* is the difference between the timestamp of the most recent event and the current timestamp. Essentially, the `IteratorAge` metric indicates how old the last processed record in the batch is. If your function is currently processing new events, then you can use the iterator age to estimate the latency between when a record is added and when your function processes it.

An increasing trend in `IteratorAge` can indicate issues with your function. For more information, see [Working with Lambda function metrics](#).

Lambda supports payloads of up to 6 MB. However, Amazon DocumentDB change stream events can be up to 16 MB in size. If your change stream tries to send Lambda a change stream event larger than 6 MB, then Lambda drops the message and emits the `OversizedRecordCount` metric. Lambda emits all metrics on a best-effort basis.

**Tutorial: Using AWS Lambda with Amazon DocumentDB Streams**

In this tutorial, you create a basic Lambda function that consumes events from an Amazon DocumentDB (with MongoDB compatibility) change stream. To complete this tutorial, you will go through the following stages:

- Set up your Amazon DocumentDB cluster, connect to it, and activate change streams on it.
- Create your Lambda function, and configure your Amazon DocumentDB cluster as an event source for your function.
- Test the end-to-end setup by inserting items into your Amazon DocumentDB database.

**Topics**
• Prerequisites

• Create the AWS Cloud9 environment

• Create the EC2 security group

• Create the secret in Secrets Manager

• Create the DocumentDB cluster

• Install the mongo shell

• Connect to the DocumentDB cluster

• Activate change streams

• Create interface VPC endpoints

• Create the execution role

• Create the Lambda function

• Create the Lambda event source mapping

• Test your function - manual invoke

• Test your function - insert a record

• Test your function - update a record

• Test your function - delete a record

• Clean up your resources

Prerequisites

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account


2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an AWS account root user is created. The root user has access to all AWS services and resources in the account. As a security best practice,
administrative access to an administrative user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing My Account.

Create an administrative user

After you sign up for an AWS account, create an administrative user so that you don’t use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing Root user and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the AWS Sign-In User Guide.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the IAM User Guide.

Create an administrative user

- For your daily administrative tasks, grant administrative access to an administrative user in AWS IAM Identity Center.

   For instructions, see Getting started in the AWS IAM Identity Center User Guide.

Sign in as the administrative user

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

   For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the AWS Sign-In User Guide.
Install the AWS Command Line Interface

If you have not yet installed the AWS Command Line Interface, follow the steps at Installing or updating the latest version of the AWS CLI to install it.

The tutorial requires a command line terminal or shell to run commands. In Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux.

Create the AWS Cloud9 environment

Before creating the Lambda function, you need to create and configure your Amazon DocumentDB cluster. The steps to set up your cluster in this tutorial is based on the procedure in Get Started with Amazon DocumentDB.
First, create an AWS Cloud9 environment. You’ll use this environment throughout this tutorial to connect to and query your DocumentDB cluster.

To create an AWS Cloud9 environment

1. Open the Cloud9 console and choose Create environment.
2. Create an environment with the following configuration:
   - Under Details:
     - Name – DocumentDBCloud9Environment
     - Environment type – New EC2 instance
   - Under New EC2 instance:
     - Instance type – t2.micro (1 GiB RAM + 1 vCPU)
     - Platform – Amazon Linux 2
     - Timeout – 30 minutes
   - Under Network settings:
     - Connection – AWS Systems Manager (SSM)
     - Expand the VPC settings dropdown.
     - Amazon Virtual Private Cloud (VPC) – Choose your default VPC.
     - Subnet – No preference
     - Keep all other default settings.
3. Choose Create. Provisioning your new AWS Cloud9 environment can take several minutes.
Next, create a EC2 security group with rules that allow traffic between your DocumentDB cluster and your Cloud9 environment.

To create an EC2 security group

2. Choose Create security group.
3. Create a security group with the following configuration:

   - Under Basic details:
     - Security group name – DocDBTutorial
     - Description – Security group for traffic between Cloud9 and DocumentDB.
     - VPC – Choose your default VPC.
   - Under Inbound rules, choose Add rule. Create a rule with the following configuration:
     - Type – Custom TCP
     - Port range – 27017
     - Source – Custom
• In the search box next to **Source**, choose the security group for the AWS Cloud9 environment you created in the previous step. To see a list of available security groups, enter cloud9 in the search box. Choose the security group with the name `aws-cloud9-<environment_name>`.

• Keep all other default settings.

4. Choose **Create security group**.

### Create the secret in Secrets Manager

To access your DocumentDB cluster manually, you must provide username and password credentials. For Lambda to access your cluster, you must provide a Secrets Manager secret that contains these same access credentials when setting up your event source mapping. In this step, you’ll create this secret.

**To create the secret in Secrets Manager**

1. Open the [Secrets Manager](#) console and choose **Store a new secret**.
2. For **Choose secret type**, choose the following options:

   • Under **Basic details**:
3. For **Configure secret**, choose the following options:
   - **Secret name** – DocumentDBSecret
     - Choose Next.

4. Choose Next.

5. Choose Store.

6. Refresh the console to verify that you successfully stored the DocumentDBSecret secret.

   Note down the **Secret ARN** of your secret. You’ll need it in a later step.

**Create the DocumentDB cluster**

In this step, you’ll create a DocumentDB cluster using the security group from the previous step.
To create a DocumentDB cluster


2. Create a cluster with the following configuration:

   • For Cluster type, choose Instance Based Cluster.

   • Under Configuration:
     • Engine version – 4.0.0
     • Instance class – db.t3.medium
     • Number of instances – 1.

   • Under Authentication:
     • Enter the Username and Password needed to connect to your cluster (same credentials as you used to create the secret in the previous step). In Confirm password, confirm your password.

     • Toggle on Show advanced settings.

     • Under Network settings:
       • Virtual Private Cloud (VPC) – Choose your default VPC.
       • Subnet group – default
       • VPC security groups – In addition to default (VPC), choose the DocDBTutorial (VPC) security group you created in the previous step.

   • Keep all other default settings.

3. Choose Create cluster. Provisioning your DocumentDB cluster can take several minutes.
Install the mongo shell

In this step, you'll install the mongo shell in your Cloud9 environment. The mongo shell is a command-line utility that you use to connect to and query your DocumentDB cluster.

To install the mongo shell on your Cloud9 environment

1. Open the Cloud9 console. Next to the DocumentDBCloud9Environment environment you created earlier, click on the Open link under the Cloud9 IDE column.

2. In the terminal window, create the MongoDB repository file with the following command:

   ```bash
   echo -e "[mongodb-org-4.0] \nname=MongoDB Repository \nbaseurl=https://repo.mongodb.org/yum/amazon/2013.03/mongodb-org/4.0/x86_64/\ngpgcheck=1 \nenabled=1 \ngpgkey=https://www.mongodb.org/static/pgp/server-4.0.asc" | sudo tee /etc/yum.repos.d/mongodb-org-4.0.repo
   ``

3. Then, install the mongo shell with the following command:

   ```bash
   sudo yum install -y mongodb-org-shell
   ```

4. To encrypt data in transit, download the public key for Amazon DocumentDB. The following command downloads a file named global-bundle.pem:
Connect to the DocumentDB cluster

You’re now ready to connect to your DocumentDB cluster using the mongo shell.

**To connect to your DocumentDB cluster**

1. Open the [DocumentDB console](https://console.aws.amazon.com/documentdb/). Under **Clusters**, choose your cluster by choosing its cluster identifier.

2. In the **Connectivity & security** tab, under **Connect to this cluster with the mongo shell**, choose **Copy**.

3. In your Cloud9 environment, paste this command into the terminal. Replace `<insertYourPassword>` with the correct password.

After entering this command, if the command prompt becomes `rs0:PRIMARY>`, then you're connected to your Amazon DocumentDB cluster.
Activate change streams

For this tutorial, you’ll track changes to the `products` collection of the `docdbdemo` database in your DocumentDB cluster. You do this by activating change streams. First, create the `docdbdemo` database and test it by inserting a record.

**To create a new database within your cluster**

1. In your Cloud9 environment, ensure that you’re still connected to your DocumentDB cluster.
2. In the terminal window, use the following command to create a new database called `docdbdemo`:

   ```
   use docdbdemo
   ```

3. Then, use the following command to insert a record into `docdbdemo`:

   ```
   db.products.insert({"hello":"world"})
   ```

   You should see output that looks like this:

   ```
   WriteResult({ "nInserted" : 1 })
   ```
4. Use the following command to list all databases:

```
show dbs
```

Ensure that your output contains the docdbdemo database:

```
docdbdemo  0.000GB
```

Next, activate change streams on the products collection of the docdbdemo database using the following command:

```
db.adminCommand({modifyChangeStreams: 1,
database: "docdbdemo",
collection: "products",
enable: true});
```

You should see output that looks like this:

```
{ "ok" : 1, "operationTime" : Timestamp(1680126165, 1) }
```
Create interface VPC endpoints

Next, create **interface VPC endpoints** to ensure that Lambda and Secrets Manager (used later to store our cluster access credentials) can connect to your default VPC.

**To create interface VPC endpoints**

1. Open the [VPC console](https://console.aws.amazon.com/vpc/). In the left menu, under **Virtual private cloud**, choose **Endpoints**.
2. Choose **Create endpoint**. Create an endpoint with the following configuration:

   - For **Name tag**, enter `lambda-default-vpc`.
   - For **Service category**, choose AWS services.
   - For **Services**, enter `lambda` in the search box. Choose the service with format `com.amazonaws.<region>.lambda`.
   - For **VPC**, choose your [default VPC](https://console.aws.amazon.com/vpc/).
   - For **Subnets**, check the boxes next to each availability zone. Choose the correct subnet ID for each availability zone.
   - For **IP address type**, select IPv4.
   - For **Security groups**, choose the default VPC security group (Group name of default), and the security group you created earlier (Group name of DocDBTutorial).
• Keep all other default settings.

• Choose Create endpoint.

3. Again, choose Create endpoint. Create an endpoint with the following configuration:

• For Name tag, enter secretsmanager-default-vpc.

• For Service category, choose AWS services.

• For Services, enter secretsmanager in the search box. Choose the service with format com.amazonaws.<region>.secretsmanager.

• For VPC, choose your default VPC.

• For Subnets, check the boxes next to each availability zone. Choose the correct subnet ID for each availability zone.

• For IP address type, select IPv4.

• For Security groups, choose the default VPC security group (Group name of default), and the security group you created earlier (Group name of DocDBTutorial).

• Keep all other default settings.

• Choose Create endpoint.

This completes the cluster setup portion of this tutorial.
Create the execution role

In the next set of steps, you'll create your Lambda function. First, you need to create the execution role that gives your function permission to access your cluster. You do this by creating an IAM policy first, then attaching this policy to an IAM role.

To create IAM policy

1. Open the [Policies page](#) in the IAM console and choose **Create policy**.
2. Choose the **JSON** tab. In the following policy, replace the Secrets Manager resource ARN in the final line of the statement with your secret ARN from earlier, and copy the policy into the editor.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "LambdaESMNetworkingAccess",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
            ],
```
3. Choose **Next: Tags**, then choose **Next: Review**.

4. For **Name**, enter `AWSDocumentDBLambdaPolicy`.

5. Choose **Create policy**.

**To create the IAM role**

1. Open the [Roles page](#) in the IAM console and choose **Create role**.

2. For **Select trusted entity**, choose the following options:
   - **Trusted entity type** – AWS service
   - **Use case** – Lambda
Choose **Next**.

3. For **Add permissions**, choose the `AWSDocumentDBLambdaPolicy` policy you just created, as well as the `AWSLambdaBasicExecutionRole` to give your function permissions to write to Amazon CloudWatch Logs.

4. Choose **Next**.

5. For **Role name**, enter `AWSDocumentDBLambdaExecutionRole`.

6. Choose **Create role**.

**Create the Lambda function**

The following example code receives a DocumentDB event input and processes the message that it contains.

```javascript
console.log('Loading function');
exports.handler = async (event, context) => {
  console.log('Received event:', JSON.stringify(event, null, 2));
  return 'OK';
};
```
To create the Lambda function

1. Copy the sample code into a file named `index.js`.
2. Create a deployment package with the following command.

   ```bash
   zip function.zip index.js
   ```

3. Use the following CLI command to create the function. Replace `us-east-1` with the region, and `123456789012` with your account ID.

   ```bash
   ```

Create the Lambda event source mapping

Create the event source mapping that associates your DocumentDB change stream with your Lambda function. After you create this event source mapping, AWS Lambda immediately starts polling the stream.

![Diagram](image-url)
To create the event source mapping

1. Open the Functions page in the Lambda console.
2. Choose the ProcessDocumentDBRecords function you created earlier.
3. Choose the Configuration tab, then choose Triggers in the left menu.
4. Choose Add trigger.
5. Under Trigger configuration, for the source, select DocumentDB.
6. Create the event source mapping with the following configuration:
   - DocumentDB cluster – Choose the cluster you created earlier.
   - Database name – docdbdemo
   - Collection name – products
   - Batch size – 1
   - Starting position – Latest
   - Authentication – BASIC_AUTH
   - Secrets Manager key – Choose the DocumentDBSecret you just created.
   - Batch window – 1
   - Full document configuration – UpdateLookup
7. Choose Add. Creating your event source mapping can take a few minutes.
To test that you created your function and event source mapping correctly, invoke your function using the `invoke` command. To do this, first copy the following event JSON into a file called `input.txt`:

```json
{
    "eventSourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:canaryclusterb2a659a2-q05tcmqkc103",
    "events": [
        {
            "event": {
                "_id": {
                    "_data": "0163eeb6e70000009010000090000000041e1"
                },
                "clusterTime": {
                    "$timestamp": {
                        "t": 1676588775,
                        "i": 9
                    }
                }
            },
            "documentKey": {
                "_id": {
```
Then, use the following command to invoke your function with this event:

```bash
aws lambda invoke --function-name ProcessDocumentDBRecords \
    --cli-binary-format raw-in-base64-out \
    --region us-east-1 \
    --payload file://input.txt out.txt
```

You should see a response that looks like the following:

```json
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

You can verify that your function successfully processed the event by checking CloudWatch Logs.

**To verify manual invocation via CloudWatch Logs**

1. Open the [Functions page](https://aws.amazon.com) in the Lambda console.
2. Choose the **Monitor** tab, then choose **View CloudWatch logs**. This takes you to the specific log group associated with your function in the CloudWatch console.
3. Choose the most recent log stream. Within the log messages, you should see the event JSON.

**Test your function - insert a record**

![Diagram of AWS Lambda workflow]

Test your end-to-end setup by interacting directly with your DocumentDB database. In the next set of steps, you'll insert a record, update it, then delete it.

**To insert a record**

1. Reconnect to your DocumentDB cluster in your Cloud9 environment.
2. Use this command to ensure that you're currently using the docdbdemo database:
   ```
   use docdbdemo
   ```
3. Insert a record into the products collection of the docdbdemo database:
   ```
   db.products.insert({"name":"Pencil", "price": 1.00})
   ```
Next, update the record you just inserted with the following command:

```javascript
db.products.update(
    { "name": "Pencil" },
    { $set: { "price": 0.50 } }
)
```

Verify that your function successfully processed this event by checking CloudWatch Logs.
Test your function - delete a record

Finally, delete the record you just updated with the following command:

```javascript
db.products.remove( { "name": "Pencil" } )
```

Verify that your function successfully processed this event by checking CloudWatch Logs.

**Clean up your resources**

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

**To delete the Lambda function**

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose Actions, Delete.
4. Type `delete` in the text input field and choose Delete.
To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose Delete.
4. Enter the name of the role in the text input field and choose Delete.

To delete the VPC endpoints

1. Open the VPC console. In the left menu, under Virtual private cloud, choose Endpoints.
2. Select the endpoints you created.
3. Choose Actions, Delete VPC endpoints.
4. Enter delete in the text input field.
5. Choose Delete.

To delete the Amazon DocumentDB cluster

1. Open the DocumentDB console.
2. Choose the DocumentDB cluster you created for this tutorial, and disable deletion protection.
3. In the main Clusters page, choose your DocumentDB cluster again.
4. Choose Actions, Delete.
5. For Create final cluster snapshot, select No.
6. Enter delete in the text input field.
7. Choose Delete.

To delete the secret in Secrets Manager

1. Open the Secrets Manager console.
2. Choose the secret you created for this tutorial.
3. Choose Actions, Delete secret.
4. Choose Schedule deletion.
To delete the Amazon EC2 security group

2. Select the security group you created for this tutorial.
3. Choose Actions, Delete security groups.
4. Choose Delete.

To delete the Cloud9 environment

1. Open the Cloud9 console.
2. Select the environment you created for this tutorial.
3. Choose Delete.
4. Enter delete in the text input field.
5. Choose Delete.
Using AWS Lambda with Amazon DynamoDB

⚠️ **Note**

If you want to send data to a target other than a Lambda function or enrich the data before sending it, see [Amazon EventBridge Pipes](#).

You can use an AWS Lambda function to process records in an [Amazon DynamoDB stream](#). With DynamoDB Streams, you can trigger a Lambda function to perform additional work each time a DynamoDB table is updated.

Lambda reads records from the stream and invokes your function **synchronously** with an event that contains stream records. Lambda reads records in batches and invokes your function to process records from the batch.

**Sections**

- [Example event](#)
- [Polling and batching streams](#)
- [Polling and stream starting positions](#)
- [Simultaneous readers of a shard in DynamoDB Streams](#)
- [Execution role permissions](#)
- [Configuring a stream as an event source](#)
- [Event source mapping APIs](#)
- [Error handling](#)
- [Amazon CloudWatch metrics](#)
- [Time windows](#)
- [Reporting batch item failures](#)
- [Amazon DynamoDB Streams configuration parameters](#)
- [Tutorial: Using AWS Lambda with Amazon DynamoDB streams](#)
- [Sample function code](#)
- [AWS SAM template for a DynamoDB application](#)
Example event

Example

```json
{
  "Records": [
    {
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "NewImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES",
        "SequenceNumber": "111",
        "SizeBytes": 26
      },
      "awsRegion": "us-west-2",
      "eventName": "INSERT",
      "eventSource": "aws:dynamodb"
    },
    {
      "eventID": "2",
      "eventVersion": "1.0",
      "dynamodb": {
        "OldImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        }
      },
      "StreamViewType": "NEW_AND_OLD_IMAGES",
      "SequenceNumber": "111",
      "SizeBytes": 26
    }
  ]
}````
Polling and batching streams

Lambda polls shards in your DynamoDB stream for records at a base rate of 4 times per second. When records are available, Lambda invokes your function and waits for the result. If processing succeeds, Lambda resumes polling until it receives more records.

By default, Lambda invokes your function as soon as records are available. If the batch that Lambda reads from the event source has only one record in it, Lambda sends only one record to the function. To avoid invoking the function with a small number of records, you can tell the event source to buffer records for up to 5 minutes by configuring a *batching window*. Before invoking the function, Lambda continues to read records from the event source until it has gathered a full batch, the batching window expires, or the batch reaches the payload limit of 6 MB. For more information, see [Batching behavior](#).
If your function returns an error, Lambda retries the batch until processing succeeds or the data expires. To avoid stalled shards, you can configure the event source mapping to retry with a smaller batch size, limit the number of retries, or discard records that are too old. To retain discarded events, you can configure the event source mapping to send details about failed batches to a standard SQS queue or standard SNS topic.

To increase concurrency, you can also process multiple batches from each shard in parallel. Lambda can process up to 10 batches in each shard simultaneously. If you increase the number of concurrent batches per shard, Lambda still ensures in-order processing at the partition-key level.

Configure the `ParallelizationFactor` setting to process one shard of a Kinesis or DynamoDB data stream with more than one Lambda invocation simultaneously. You can specify the number of concurrent batches that Lambda polls from a shard via a parallelization factor from 1 (default) to 10. For example, when you set `ParallelizationFactor` to 2, you can have 200 concurrent Lambda invocations at maximum to process 100 Kinesis data shards (though in practice, you may see a different values for the `ConcurrentExecutions` metric). This helps scale up the processing throughput when the data volume is volatile and the `IteratorAge` is high. Note that parallelization factor will not work if you are using Kinesis aggregation. For more information, see New AWS Lambda scaling controls for Kinesis and DynamoDB event sources. Also, see the Serverless Data Processing on AWS workshop for complete tutorials.

### Polling and stream starting positions

Be aware that stream polling during event source mapping creation and updates is eventually consistent.

- During event source mapping creation, it may take several minutes to start polling events from the stream.
- During event source mapping updates, it may take several minutes to stop and restart polling events from the stream.

This behavior means that if you specify LATEST as the starting position for the stream, the event source mapping could miss events during creation or updates. To ensure that no events are missed, specify the stream starting position as TRIM_HORIZON.
Simultaneous readers of a shard in DynamoDB Streams

For single-Region tables that are not global tables, you can design for up to two Lambda functions to read from the same DynamoDB Streams shard at the same time. Exceeding this limit can result in request throttling. For global tables, we recommend you limit the number of simultaneous functions to one to avoid request throttling.

Execution role permissions

Lambda needs the following permissions to manage resources related to your DynamoDB stream. Add them to your function’s execution role.

- `dynamodb:DescribeStream`
- `dynamodb:GetRecords`
- `dynamodb:GetShardIterator`
- `dynamodb:ListStreams`

The `AWSLambdaDynamoDBExecutionRole` managed policy includes these permissions. For more information, see Lambda execution role.

To send records of failed batches to a standard SQS queue or standard SNS topic, your function needs additional permissions. Each destination service requires a different permission, as follows:

- **Amazon SQS** – `sqs:SendMessage`
- **Amazon SNS** – `sns:Publish`

Configuring a stream as an event source

Create an event source mapping to tell Lambda to send records from your stream to a Lambda function. You can create multiple event source mappings to process the same data with multiple Lambda functions, or to process items from multiple streams with a single function.

To configure your function to read from DynamoDB Streams in the Lambda console, create a DynamoDB trigger.

To create a trigger

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
4. Choose a trigger type.
5. Configure the required options, and then choose Add.

Lambda supports the following options for DynamoDB event sources.

**Event source options**

- **DynamoDB table** – The DynamoDB table to read records from.
- **Batch size** – The number of records to send to the function in each batch, up to 10,000. Lambda passes all of the records in the batch to the function in a single call, as long as the total size of the events doesn't exceed the payload limit for synchronous invocation (6 MB).
- **Batch window** – Specify the maximum amount of time to gather records before invoking the function, in seconds.
- **Starting position** – Process only new records, or all existing records.
  - **Latest** – Process new records that are added to the stream.
  - **Trim horizon** – Process all records in the stream.

After processing any existing records, the function is caught up and continues to process new records.

- **On-failure destination** – A standard SQS queue or standard SNS topic for records that can't be processed. When Lambda discards a batch of records that's too old or has exhausted all retries, Lambda sends details about the batch to the queue or topic.
- **Retry attempts** – The maximum number of times that Lambda retries when the function returns an error. This doesn't apply to service errors or throttles where the batch didn't reach the function.
- **Maximum age of record** – The maximum age of a record that Lambda sends to your function.
- **Split batch on error** – When the function returns an error, split the batch into two before retrying. Your original batch size setting remains unchanged.
- **Concurrent batches per shard** – Concurrently process multiple batches from the same shard.
- **Enabled** – Set to true to enable the event source mapping. Set to false to stop processing records. Lambda keeps track of the last record processed and resumes processing from that point when the mapping is reenabled.
Note
You are not charged for GetRecords API calls invoked by Lambda as part of DynamoDB triggers.

To manage the event source configuration later, choose the trigger in the designer.

Event source mapping APIs

To manage an event source with the AWS Command Line Interface (AWS CLI) or an AWS SDK, you can use the following API operations:

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

The following example uses the AWS CLI to map a function named my-function to a DynamoDB stream that its Amazon Resource Name (ARN) specifies, with a batch size of 500.

```
aws lambda create-event-source-mapping --function-name my-function --batch-size 500 --
maximum-batching-window-in-seconds 5 --starting-position LATEST \
stream/2019-06-10T19:26:16.525
```

You should see the following output:

```
{
    "UUID": "14e0db71-5d35-4eb5-b481-8945cf9d10c2",
    "BatchSize": 500,
    "MaximumBatchingWindowInSeconds": 5,
    "ParallelizationFactor": 1,
    "LastModified": 1560209851.963,
}
```
Configure additional options to customize how batches are processed and to specify when to discard records that can't be processed. The following example updates an event source mapping to send a failure record to a standard SQS queue after two retry attempts, or if the records are more than an hour old.

```bash
aws lambda update-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b --maximum-retry-attempts 2 --maximum-record-age-in-seconds 3600 --destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-east-2:123456789012:dlq"}}'
```

You should see this output:

```
{
    "UUID": "f89f8514-cdd9-4602-9e1f-01a5b77d449b",
    "BatchSize": 100,
    "MaximumBatchingWindowInSeconds": 0,
    "ParallelizationFactor": 1,
    "LastModified": 1573243620.0,
    "LastProcessingResult": "PROBLEM: Function call failed",
    "State": "Updating",
    "StateTransitionReason": "User action",
    "DestinationConfig": {},
    "MaximumRecordAgeInSeconds": 604800,
    "BisectBatchOnFunctionError": false,
    "MaximumRetryAttempts": 10000
}
```

Updated settings are applied asynchronously and aren't reflected in the output until the process completes. Use the `get-event-source-mapping` command to view the current status.
You should see this output:

```json
{
    "UUID": "f89f8514-cdd9-4602-9e1f-01a5b77d449b",
    "BatchSize": 100,
    "MaximumBatchingWindowInSeconds": 0,
    "ParallelizationFactor": 1,
    "LastModified": 1573244760.0,
    "LastProcessingResult": "PROBLEM: Function call failed",
    "State": "Enabled",
    "StateTransitionReason": "User action",
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "arn:aws:sqs:us-east-2:123456789012:dlq"
        }
    },
    "MaximumRecordAgeInSeconds": 3600,
    "BisectBatchOnFunctionError": false,
    "MaximumRetryAttempts": 2
}
```

To process multiple batches concurrently, use the `--parallelization-factor` option.

```bash
aws lambda update-event-source-mapping --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284 --parallelization-factor 5
```

## Error handling

The event source mapping that reads records from your DynamoDB stream, invokes your function synchronously, and retries on errors. If Lambda throttles the function or returns an error without invoking the function, Lambda retries until the records expire or exceed the maximum age that you configure on the event source mapping.

If the function receives the records but returns an error, Lambda retries until the records in the batch expire, exceed the maximum age, or reach the configured retry quota. For function errors,
you can also configure the event source mapping to split a failed batch into two batches. Retrying
with smaller batches isolates bad records and works around timeout issues. Splitting a batch does
not count towards the retry quota.

If the error handling measures fail, Lambda discards the records and continues processing batches
from the stream. With the default settings, this means that a bad record can block processing on
the affected shard for up to one day. To avoid this, configure your function's event source mapping
with a reasonable number of retries and a maximum record age that fits your use case.

To retain a record of discarded batches, configure a failed-event destination. Lambda sends a
document to the destination queue or topic with details about the batch.

To configure a destination for failed-event records

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Under Function overview, choose Add destination.
4. For Source, choose Stream invocation.
5. For Stream, choose a stream that is mapped to the function.
6. For Destination type, choose the type of resource that receives the invocation record.
7. For Destination, choose a resource.
8. Choose Save.

The following example shows an invocation record for a DynamoDB stream.

Example Invocation Record

```json
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    }
}
```
You can use this information to retrieve the affected records from the stream for troubleshooting. The actual records aren't included, so you must process this record and retrieve them from the stream before they expire and are lost.

**Amazon CloudWatch metrics**

Lambda emits the `IteratorAge` metric when your function finishes processing a batch of records. The metric indicates how old the last record in the batch was when processing finished. If your function is processing new events, you can use the iterator age to estimate the latency between when a record is added and when the function processes it.

An increasing trend in iterator age can indicate issues with your function. For more information, see [Working with Lambda function metrics](#).

**Time windows**

Lambda functions can run continuous stream processing applications. A stream represents unbounded data that flows continuously through your application. To analyze information from this continuously updating input, you can bound the included records using a window defined in terms of time.

Tumbling windows are distinct time windows that open and close at regular intervals. By default, Lambda invocations are stateless—you cannot use them for processing data across multiple continuous invocations without an external database. However, with tumbling windows, you can maintain your state across invocations. This state contains the aggregate result of the messages
previously processed for the current window. Your state can be a maximum of 1 MB per shard. If it exceeds that size, Lambda terminates the window early.

Each record in a stream belongs to a specific window. Lambda will process each record at least once, but doesn’t guarantee that each record will be processed only once. In rare cases, such as error handling, some records might be processed more than once. Records are always processed in order the first time. If records are processed more than once, they might be processed out of order.

**Aggregation and processing**

Your user managed function is invoked both for aggregation and for processing the final results of that aggregation. Lambda aggregates all records received in the window. You can receive these records in multiple batches, each as a separate invocation. Each invocation receives a state. Thus, when using tumbling windows, your Lambda function response must contain a `state` property. If the response does not contain a `state` property, Lambda considers this a failed invocation. To satisfy this condition, your function can return a `TimeWindowEventResponse` object, which has the following JSON shape:

**Example TimeWindowEventResponse values**

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

**Note**

For Java functions, we recommend using a `Map<String, String>` to represent the state.

At the end of the window, the flag `isFinalInvokeForWindow` is set to `true` to indicate that this is the final state and that it’s ready for processing. After processing, the window completes and your final invocation completes, and then the state is dropped.

At the end of your window, Lambda uses final processing for actions on the aggregation results. Your final processing is synchronously invoked. After successful invocation, your function
checkpoints the sequence number and stream processing continues. If invocation is unsuccessful, your Lambda function suspends further processing until a successful invocation.

**Example DynamodbTimeWindowEvent**

```json
{
   "Records": [
      {
        "eventID": "1",
        "eventName": "INSERT",
        "eventVersion": "1.0",
        "eventSource": "aws:dynamodb",
        "awsRegion": "us-east-1",
        "dynamodb": {
          "Keys": {
            "Id": {
              "N": "101"
            }
          },
          "NewImage": {
            "Message": {
              "S": "New item!"
            },
            "Id": {
              "N": "101"
            }
          },
          "SequenceNumber": "111",
          "SizeBytes": 26,
          "StreamViewType": "NEW_AND_OLD_IMAGES"
        },
        "eventSourceARN": "stream-ARN"
      },
      {
        "eventID": "2",
        "eventName": "MODIFY",
        "eventVersion": "1.0",
        "eventSource": "aws:dynamodb",
        "awsRegion": "us-east-1",
        "dynamodb": {
          "Keys": {
            "Id": {
              "N": "101"
            }
          }
        }
      }
   ]
}```
{ "eventID":"3", "eventName":"REMOVE", "eventVersion":"1.0", "eventSource":"aws:dynamodb", "awsRegion":"us-east-1", "dynamodb": { "Keys": { "Id": { "N":"101" } }, "OldImage": { "Message": { "S":"This item has changed" }, "Id": { "N":"101" } }, "SequenceNumber":"222", "SizeBytes":59, "StreamViewType":"NEW_AND_OLD_IMAGES" }, "eventSourceARN":"stream-ARN" },

Time windows
Configuration

You can configure tumbling windows when you create or update an event source mapping. To configure a tumbling window, specify the window in seconds. The following example AWS Command Line Interface (AWS CLI) command creates a streaming event source mapping that has a tumbling window of 120 seconds. The Lambda function defined for aggregation and processing is named tumbling-window-example-function.

```
```

Lambda determines tumbling window boundaries based on the time when records were inserted into the stream. All records have an approximate timestamp available that Lambda uses in boundary determinations.

Tumbling window aggregations do not support resharding. When the shard ends, Lambda considers the window closed, and the child shards start their own window in a fresh state.

Tumbling windows fully support the existing retry policies maxRetryAttempts and maxRecordAge.
Example Handler.py – Aggregation and processing

The following Python function demonstrates how to aggregate and then process your final state:

```python
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

    # Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

    # Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    # Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['dynamodb']['NewImage']['Id']] = state.get(record['dynamodb']['NewImage']['Id'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

Reporting batch item failures

When consuming and processing streaming data from an event source, by default Lambda checkpoints to the highest sequence number of a batch only when the batch is a complete success. Lambda treats all other results as a complete failure and retries processing the batch up to the retry limit. To allow for partial successes while processing batches from a stream, turn on ReportBatchItemFailures. Allowing partial successes can help to reduce the number of retries on a record, though it doesn’t entirely prevent the possibility of retries in a successful record.

To turn on ReportBatchItemFailures, include the enum value `ReportBatchItemFailures` in the FunctionResponseTypes list. This list indicates which response types are enabled for your function. You can configure this list when you create or update an [event source mapping](https://docs.aws.amazon.com/lambda/latest/dg/使用Lambda的事件源映射.html).
Report syntax

When configuring reporting on batch item failures, the StreamsEventResponse class is returned with a list of batch item failures. You can use a StreamsEventResponse object to return the sequence number of the first failed record in the batch. You can also create your own custom class using the correct response syntax. The following JSON structure shows the required response syntax:

```json
{
  "batchItemFailures": [
    {
      "itemIdentifier": "<id>
    }
  ]
}
```

Note

If the batchItemFailures array contains multiple items, Lambda uses the record with the lowest sequence number as the checkpoint. Lambda then retries all records starting from that checkpoint.

Success and failure conditions

Lambda treats a batch as a complete success if you return any of the following:

- An empty batchItemFailure list
- A null batchItemFailure list
- An empty EventResponse
- A null EventResponse

Lambda treats a batch as a complete failure if you return any of the following:

- An empty string itemIdentifier
- A null itemIdentifier
- An itemIdentifier with a bad key name
Lambda retries failures based on your retry strategy.

**Bisecting a batch**

If your invocation fails and `BisectBatchOnFunctionError` is turned on, the batch is bisected regardless of your `ReportBatchItemFailures` setting.

When a partial batch success response is received and both `BisectBatchOnFunctionError` and `ReportBatchItemFailures` are turned on, the batch is bisected at the returned sequence number and Lambda retries only the remaining records.

**Java**

**Example Handler.java – return new StreamsEventResponse()**

```java
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent, Serializable> {

    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord : input.getRecords()) {
            try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();
            } catch (Exception e) {
                batchItemFailures.add(new StreamsEventResponse.BatchItemFailure(dynamodbStreamRecord.batchItem.getRecord(), e.getMessage(), curRecordSequenceNumber));
            }
        }

        return new StreamsEventResponse(input.getRecordsSequenceNumber(), batchItemFailures);
    }
}
```
} catch (Exception e) {
    /* Since we are working with streams, we can return the failed item immediately. 
    Lambda will immediately begin to retry processing from this failed item onwards. */
    batchItemFailures.add(new StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
    return new StreamsEventResponse(batchItemFailures);
}

return new StreamsEventResponse();
}

Python

Example Handler.py – return batchItemFailures

def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {
                "batchItemFailures": [{"itemIdentifier": curRecordSequenceNumber}]
            }

        return {
            "batchItemFailures": []
        }

Amazon DynamoDB Streams configuration parameters

All Lambda event source types share the same CreateEventSourceMapping and UpdateEventSourceMapping API operations. However, only some of the parameters apply to DynamoDB Streams.
### Event source parameters that apply to DynamoDB Streams

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Default</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BatchSize</td>
<td>N</td>
<td>100</td>
<td>Maximum: 10,000</td>
</tr>
<tr>
<td>BisectBatchOnFunctionError</td>
<td>N</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>DestinationConfig</td>
<td>N</td>
<td></td>
<td>Standard Amazon SQS queue or standard Amazon SNS topic destination for discarded records</td>
</tr>
<tr>
<td>Enabled</td>
<td>N</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>EventSourceArn</td>
<td>Y</td>
<td></td>
<td>ARN of the data stream or a stream consumer</td>
</tr>
<tr>
<td>FilterCriteria</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FunctionName</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaximumBatchingWindowInSeconds</td>
<td>N</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>MaximumRecordAgeInSeconds</td>
<td>N</td>
<td>-1</td>
<td>-1 means infinite: failed records are retried until the record expires</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Minimum: 60</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Maximum: 604,800</td>
</tr>
<tr>
<td>MaximumRetryAttempts</td>
<td>N</td>
<td>-1</td>
<td>-1 means infinite: failed records are</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Required</td>
<td>Default</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------</td>
<td>---------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>retried until the record expires</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Minimum: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Maximum: 10,000</td>
</tr>
<tr>
<td>ParallelizationFactor</td>
<td>N</td>
<td>1</td>
<td>Maximum: 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StartingPosition</td>
<td>Y</td>
<td></td>
<td>TRIM_HORIZON or LATEST</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TumblingWindowInSeconds</td>
<td>N</td>
<td></td>
<td>Minimum: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Maximum: 900</td>
</tr>
</tbody>
</table>

**Tutorial: Using AWS Lambda with Amazon DynamoDB streams**

In this tutorial, you create a Lambda function to consume events from an Amazon DynamoDB stream.

**Prerequisites**

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console](#) to create your first Lambda function.

To complete the following steps, you need the [AWS Command Line Interface (AWS CLI) version 2](#). Commands and the expected output are listed in separate blocks:

```bash
aws --version
```

You should see the following output:

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

For long commands, an escape character (\) is used to split a command over multiple lines.
On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux. Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

Create the execution role

Create the execution role that gives your function permission to access AWS resources.

To create an execution role

1. Open the roles page in the IAM console.
2. Choose Create role.
3. Create a role with the following properties.
   - **Trusted entity** – Lambda.
   - **Permissions** – AWSLambdaDynamoDBExecutionRole.
   - **Role name** – `lambda-dynamodb-role`.

The AWSLambdaDynamoDBExecutionRole has the permissions that the function needs to read items from DynamoDB and write logs to CloudWatch Logs.

Create the function

The following example code receives a DynamoDB event input and processes the messages that it contains. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Note

For sample code in other languages, see Sample function code.
Example index.mjs

```javascript
console.log('Loading function');

export const handler = (event, context, callback) => {
  console.log(JSON.stringify(event, null, 2));
  event.Records.forEach(record => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
  });
  callback(null, "message" );
};
```

To create the function

1. Copy the sample code into a file named index.mjs.
2. Create a deployment package.

```
zip function.zip index.mjs
```

3. Create a Lambda function with the create-function command.

```
aws lambda create-function --function-name ProcessDynamoDBRecords \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--role arn:aws:iam::111122223333:role/lambda-dynamodb-role
```

Test the Lambda function

In this step, you invoke your Lambda function manually using the invoke AWS Lambda CLI command and the following sample DynamoDB event.

Example input.txt

```json
{
    "Records": [
        {
            "eventID":"1",
            "eventName":"INSERT",
            "eventVersion":"1.0",
            "eventSource":"aws:dynamodb",
```
"awsRegion":"us-east-1",
"dynamodb":{
  "Keys":{
    "Id":{
      "N":"101"
    }
  },
  "NewImage":{
    "Message":{
      "S":"New item!"
    },
    "Id":{
      "N":"101"
    }
  },
  "SequenceNumber":"111",
  "SizeBytes":26,
  "StreamViewType":"NEW_AND_OLD_IMAGES"
},
"eventSourceARN":"stream-ARN"
},
{
  "eventID":"2",
  "eventName":"MODIFY",
  "eventVersion":"1.0",
  "eventSource":"aws:dynamodb",
  "awsRegion":"us-east-1",
  "dynamodb":{
    "Keys":{
      "Id":{
        "N":"101"
      }
    },
    "NewImage":{
      "Message":{
        "S":"This item has changed"
      },
      "Id":{
        "N":"101"
      }
    },
    "OldImage":{
      "Message":{
        "S":"New item!"
      }
    }
  }
}
Run the following `invoke` command.

```bash
aws lambda invoke --function-name ProcessDynamoDBRecords
```
The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the AWS Command Line Interface User Guide for Version 2.

The function returns the string message in the response body.

Verify the output in the `outputfile.txt` file.

Create a DynamoDB table with a stream enabled

Create an Amazon DynamoDB table with a stream enabled.

To create a DynamoDB table

1. Open the DynamoDB console.
2. Choose Create table.
3. Create a table with the following settings.
   • **Table name** – `lambda-dynamodb-stream`
   • **Primary key** – `id` (string)
4. Choose Create.

To enable streams

1. Open the DynamoDB console.
2. Choose Tables.
3. Choose the `lambda-dynamodb-stream` table.
4. Under Exports and streams, choose DynamoDB stream details.
5. Choose Enable.
6. Choose Enable stream.
Write down the stream ARN. You need this in the next step when you associate the stream with your Lambda function. For more information on enabling streams, see Capturing table activity with DynamoDB Streams.

**Add an event source in AWS Lambda**

Create an event source mapping in AWS Lambda. This event source mapping associates the DynamoDB stream with your Lambda function. After you create this event source mapping, AWS Lambda starts polling the stream.

Run the following AWS CLI `create-event-source-mapping` command. After the command runs, note down the UUID. You'll need this UUID to refer to the event source mapping in any commands, for example, when deleting the event source mapping.

```bash
aws lambda create-event-source-mapping --function-name ProcessDynamoDBRecords --batch-size 100 --starting-position LATEST --event-source DynamoDB-stream-arn
```

This creates a mapping between the specified DynamoDB stream and the Lambda function. You can associate a DynamoDB stream with multiple Lambda functions, and associate the same Lambda function with multiple streams. However, the Lambda functions will share the read throughput for the stream they share.

You can get the list of event source mappings by running the following command.

```bash
aws lambda list-event-source-mappings
```

The list returns all of the event source mappings you created, and for each mapping it shows the `LastProcessingResult`, among other things. This field is used to provide an informative message if there are any problems. Values such as `No records processed` (indicates that AWS Lambda has not started polling or that there are no records in the stream) and `OK` (indicates AWS Lambda successfully read records from the stream and invoked your Lambda function) indicate that there are no issues. If there are issues, you receive an error message.

If you have a lot of event source mappings, use the function name parameter to narrow down the results.

```bash
aws lambda list-event-source-mappings --function-name ProcessDynamoDBRecords
```
Test the setup

Test the end-to-end experience. As you perform table updates, DynamoDB writes event records to the stream. As AWS Lambda polls the stream, it detects new records in the stream and invokes your Lambda function on your behalf by passing events to the function.

1. In the DynamoDB console, add, update, and delete items to the table. DynamoDB writes records of these actions to the stream.
2. AWS Lambda polls the stream and when it detects updates to the stream, it invokes your Lambda function by passing in the event data it finds in the stream.
3. Your function runs and creates logs in Amazon CloudWatch. You can verify the logs reported in the Amazon CloudWatch console.

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose Actions, Delete.
4. Type delete in the text input field and choose Delete.

To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose Delete.
4. Enter the name of the role in the text input field and choose Delete.

To delete the DynamoDB table

1. Open the Tables page of the DynamoDB console.
2. Select the table you created.
3. Choose **Delete**.
4. Enter **delete** in the text box.
5. Choose **Delete table**.

### Sample function code

Sample code is available for the following languages.

**Topics**

- [Node.js](#)
- [Java 11](#)
- [C#](#)
- [Python 3](#)
- [Go](#)

#### Node.js

The following example processes messages from DynamoDB, and logs their contents.

**Example ProcessDynamoDBStream.js**

```javascript
console.log('Loading function');

exports.lambda_handler = function(event, context, callback) {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        console.log(record.eventID);
        console.log(record.eventName);
        console.log('DynamoDB Record: %j', record.dynamodb);
    });
    callback(null, "message");
};
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Node.js Lambda functions with .zip file archives](#).
Java 11

The following example processes messages from DynamoDB, and logs their contents. `handleRequest` is the handler that AWS Lambda invokes and provides event data. The handler uses the predefined `DynamodbEvent` class, which is defined in the `aws-lambda-java-events` library.

**Example DDBEventProcessor.java**

```java
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler2;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;

public class DDBEventProcessor implements RequestHandler2<DynamodbEvent, String> {
    public String handleRequest(DynamodbEvent ddbEvent, Context context) {
        for (DynamodbStreamRecord record : ddbEvent.getRecords()){
            System.out.println(record.getEventID());
            System.out.println(record.getEventName());
            System.out.println(record.getDynamodb().toString());
        }
        return "Successfully processed " + ddbEvent.getRecords().size() + " records.";
    }
}
```

If the handler returns normally without exceptions, Lambda considers the input batch of records as processed successfully and begins reading new records in the stream. If the handler throws an exception, Lambda considers the input batch of records as not processed and invokes the function with the same batch of records again.

**Dependencies**

- `aws-lambda-java-core`
- `aws-lambda-java-events`
Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [Deploy Java Lambda functions with .zip or JAR file archives](#).

**C#**

The following example processes messages from DynamoDB, and logs their contents. ProcessDynamoEvent is the handler that AWS Lambda invokes and provides event data. The handler uses the predefined DynamoDbEvent class, which is defined in the Amazon.Lambda.DynamoDBEvents library.

**Example ProcessingDynamoDBStreams.cs**

```csharp
using System;
using System.IO;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;
using Amazon.Lambda.Serialization.Json;

namespace DynamoDBStreams
{
    public class DdbSample
    {
        private static readonly JsonSerializer _jsonSerializer = new JsonSerializer();

        public void ProcessDynamoEvent(DynamoDBEvent dynamoEvent)
        {
            Console.WriteLine($"Beginning to process {dynamoEvent.Records.Count} records..."");

            foreach (var record in dynamoEvent.Records)
            {
                Console.WriteLine($"Event ID: {record.EventID}");
                Console.WriteLine($"Event Name: {record.EventName}"");

                string streamRecordJson = SerializeObject(record.Dynamodb);
                Console.WriteLine($"DynamoDB Record:");
                Console.WriteLine(streamRecordJson);
            }

            Console.WriteLine("Stream processing complete.");
        }
    }
}
```

Sample code
private string SerializeObject(object streamRecord)
{
    using (var ms = new MemoryStream())
    {
        _jsonSerializer.Serialize(streamRecord, ms);
        return Encoding.UTF8.GetString(ms.ToArray());
    }
}

Replace the Program.cs in a .NET Core project with the above sample. For instructions, see Build and deploy C# Lambda functions with .zip file archives.

Python 3

The following example processes messages from DynamoDB, and logs their contents.

Example ProcessDynamoDBStream.py

```python
from __future__ import print_function

def lambda_handler(event, context):
    for record in event['Records']:
        print(record['eventID'])
        print(record['eventName'])
        print('Successfully processed %s records.' % str(len(event['Records'])))
```

Zip up the sample code to create a deployment package. For instructions, see Working with .zip file archives for Python Lambda functions.

Go

The following example processes messages from DynamoDB, and logs their contents.

Example

```go
import {
    "strings"
    "github.com/aws/aws-lambda-go/events"
```
func handleRequest(ctx context.Context, e events.DynamoDBEvent) {
    for _, record := range e.Records {
        fmt.Printf("Processing request data for event ID %s, type %s.\n", record.EventID, record.EventName)
        // Print new values for attributes of type String
        for name, value := range record.Change.NewImage {
            if value.DataType() == events.DataTypeString {
                fmt.Printf("Attribute name: %s, value: %s\n", name, value.String())
            }
        }
    }
}

Build the executable with `go build` and create a deployment package. For instructions, see [Deploy Go Lambda functions with .zip file archives](https://docs.aws.amazon.com/lambda/latest/dg/func-deploy-go.html).

### AWS SAM template for a DynamoDB application

You can build this application using [AWS SAM](https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/). To learn more about creating AWS SAM templates, see [AWS SAM template basics](https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/) in the [AWS Serverless Application Model Developer Guide](https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/).

Below is a sample AWS SAM template for the tutorial application. Copy the text below to a .yaml file and save it next to the ZIP package you created previously. Note that the Handler and Runtime parameter values should match the ones you used when you created the function in the previous section.

**Example template.yaml**

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  ProcessDynamoDBStream:
    Type: AWS::Serverless::Function
    Properties:
      Handler: handler
      Runtime: runtime
      Policies: AWSLambdaDynamoDBExecutionRole
      Events:
```

Stream:
  Type: DynamoDB
Properties:
  Stream: !GetAtt DynamoDBTable.StreamArn
  BatchSize: 100
  StartingPosition: TRIM_HORIZON

DynamoDBTable:
  Type: AWS::DynamoDB::Table
Properties:
  AttributeDefinitions:
    - AttributeName: id
      AttributeType: S
  KeySchema:
    - AttributeName: id
      KeyType: HASH
  ProvisionedThroughput:
    ReadCapacityUnits: 5
    WriteCapacityUnits: 5
  StreamSpecification:
    StreamViewType: NEW_IMAGE

For information on how to package and deploy your serverless application using the package and deploy commands, see Deploying serverless applications in the AWS Serverless Application Model Developer Guide.
Using AWS Lambda with Amazon EC2

You can use AWS Lambda to process lifecycle events from Amazon Elastic Compute Cloud and manage Amazon EC2 resources. Amazon EC2 sends events to Amazon EventBridge (CloudWatch Events) for lifecycle events such as when an instance changes state, when an Amazon Elastic Block Store volume snapshot completes, or when a spot instance is scheduled to be terminated. You configure EventBridge (CloudWatch Events) to forward those events to a Lambda function for processing.

EventBridge (CloudWatch Events) invokes your Lambda function asynchronously with the event document from Amazon EC2.

Example instance lifecycle event

```json
{
    "version": "0",
    "id": "b6ba298a-7732-2226-xmpl-976312c1a050",
    "detail-type": "EC2 Instance State-change Notification",
    "source": "aws.ec2",
    "account": "111122223333",
    "time": "2019-10-02T17:59:30Z",
    "region": "us-east-1",
    "resources": ["arn:aws:ec2:us-east-1:111122223333:instance/i-0c314xmplcd5b8173"],
    "detail": {
        "instance-id": "i-0c314xmplcd5b8173",
        "state": "running"
    }
}
```

For details on configuring events in EventBridge (CloudWatch Events), see Using AWS Lambda with Amazon EventBridge (CloudWatch Events). For an example function that processes Amazon EBS snapshot notifications, see Amazon EventBridge (CloudWatch Events) for Amazon EBS in the Amazon EC2 User Guide for Linux Instances.

You can also use the AWS SDK to manage instances and other resources with the Amazon EC2 API.
Permissions

To process lifecycle events from Amazon EC2, EventBridge (CloudWatch Events) needs permission to invoke your function. This permission comes from the function's resource-based policy. If you use the EventBridge (CloudWatch Events) console to configure an event trigger, the console updates the resource-based policy on your behalf. Otherwise, add a statement like the following:

Example resource-based policy statement for Amazon EC2 lifecycle notifications

```json
{
    "Sid": "ec2-events",
    "Effect": "Allow",
    "Principal": {
        "Service": "events.amazonaws.com"
    },
    "Action": "lambda:InvokeFunction",
    "Condition": {
        "ArnLike": {
            "AWS:SourceArn": "arn:aws:events:us-east-1:12456789012:rule/*"
        }
    }
}
```

To add a statement, use the add-permission AWS CLI command.

```bash
```

If your function uses the AWS SDK to manage Amazon EC2 resources, add Amazon EC2 permissions to the function's execution role.
Tutorial: Configuring a Lambda function to access Amazon ElastiCache in an Amazon VPC

To learn how to configure Lambda to access Amazon ElastiCache in an Amazon VPC, see the Lambda tutorial in the ElastiCache for Memcached User Guide.
Using AWS Lambda with an Application Load Balancer

You can use a Lambda function to process requests from an Application Load Balancer. Elastic Load Balancing supports Lambda functions as a target for an Application Load Balancer. Use load balancer rules to route HTTP requests to a function, based on path or header values. Process the request and return an HTTP response from your Lambda function.

Elastic Load Balancing invokes your Lambda function synchronously with an event that contains the request body and metadata.

Example Application Load Balancer request event

```json
{
   "requestContext": {
      "elb": {
         "targetGroupArn": "arn:aws:elasticloadbalancing:us-east-1:123456789012:targetgroup/lambda-279XGJDqGZ5rsrHC2Fjr/49e9d65c45c6791a"
      },
      "httpMethod": "GET",
      "path": "/lambda",
      "queryStringParameters": {
         "query": "1234ABCD"
      },
      "headers": {
         "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8",
         "accept-encoding": "gzip",
         "accept-language": "en-US,en;q=0.9",
         "connection": "keep-alive",
         "host": "lambda-alb-123578498.us-east-1.elb.amazonaws.com",
         "upgrade-insecure-requests": "1",
         "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36",
         "x-amzn-trace-id": "Root=1-5c536348-3d683b8b04734faae651f476",
         "x-forwarded-for": "72.12.164.125",
         "x-forwarded-port": "80",
         "x-forwarded-proto": "http",
         "x-imforwards": "20"
      },
      "body": "",
      "isBase64Encoded": false
   }
}```
Your function processes the event and returns a response document to the load balancer in JSON. Elastic Load Balancing converts the document to an HTTP success or error response and returns it to the user.

**Example response document format**

```
{
    "statusCode": 200,
    "statusDescription": "200 OK",
    "isBase64Encoded": false,
    "headers": {
        "Content-Type": "text/html"
    },
    "body": "<h1>Hello from Lambda!</h1>"
}
```

To configure an Application Load Balancer as a function trigger, grant Elastic Load Balancing permission to run the function, create a target group that routes requests to the function, and add a rule to the load balancer that sends requests to the target group.

Use the `aws lambda add-permission` command to add a permission statement to your function's resource-based policy.

```
aws lambda add-permission --function-name alb-function \
--statement-id load-balancer --action "lambda:InvokeFunction" \
--principal elasticloadbalancing.amazonaws.com
```

You should see the following output:

```
{
```

For instructions on configuring the Application Load Balancer listener and target group, see [Lambda functions as a target](https://docs.aws.amazon.com/elasticloadbalancing/latest/application/lambda-functions.html) in the *User Guide for Application Load Balancers*. 

Using Amazon EFS with Lambda

Lambda integrates with Amazon Elastic File System (Amazon EFS) to support secure, shared file system access for Lambda applications. You can configure functions to mount a file system during initialization with the NFS protocol over the local network within a VPC. Lambda manages the connection and encrypts all traffic to and from the file system.

The file system and the Lambda function must be in the same region. A Lambda function in one account can mount a file system in a different account. For this scenario, you configure VPC peering between the function VPC and the file system VPC.

Note
To configure a function to connect to a file system, see Configuring file system access for Lambda functions.

Amazon EFS supports file locking to prevent corruption if multiple functions try to write to the same file system at the same time. Locking in Amazon EFS follows the NFS v4.1 protocol for advisory locking, and enables your applications to use both whole file and byte range locks.

Amazon EFS provides options to customize your file system based on your application's need to maintain high performance at scale. There are three primary factors to consider: the number of connections, throughput (in MiB per second), and IOPS.

Quotas
For detail on file system quotas and limits, see Quotas for Amazon EFS file systems in the Amazon Elastic File System User Guide.

To avoid issues with scaling, throughput, and IOPS, monitor the metrics that Amazon EFS sends to Amazon CloudWatch. For an overview of monitoring in Amazon EFS, see Monitoring Amazon EFS in the Amazon Elastic File System User Guide.

Sections
- Connections
- Throughput
Connections

Amazon EFS supports up to 25,000 connections per file system. During initialization, each instance of a function creates a single connection to its file system that persists across invocations. This means that you can reach 25,000 concurrency across one or more functions connected to a file system. To limit the number of connections a function creates, use reserved concurrency.

However, when you make changes to your function's code or configuration at scale, there is a temporary increase in the number of function instances beyond the current concurrency. Lambda provisions new instances to handle new requests and there is some delay before old instances close their connections to the file system. To avoid hitting the maximum connections limit during a deployment, use rolling deployments. With rolling deployments, you gradually shift traffic to the new version each time you make a change.

If you connect to the same file system from other services such as Amazon EC2, you should also be aware of the scaling behavior of connections in Amazon EFS. A file system supports the creation of up to 3,000 connections in a burst, after which it supports 500 new connections per minute.

To monitor and trigger an alarm on connections, use the ClientConnections metric.

Throughput

At scale, it is also possible to exceed the maximum throughput for a file system. In bursting mode (the default), a file system has a low baseline throughput that scales linearly with its size. To allow for bursts of activity, the file system is granted burst credits that allow it to use 100 MiB/s or more of throughput. Credits accumulate continually and are expended with every read and write operation. If the file system runs out of credits, it throttles read and write operations beyond the baseline throughput, which can cause invocations to time out.

Note

If you use provisioned concurrency, your function can consume burst credits even when idle. With provisioned concurrency, Lambda initializes instances of your function before it is invoked, and recycles instances every few hours. If you use files on an attached file system during initialization, this activity can use all of your burst credits.
To monitor and trigger an alarm on throughput, use the BurstCreditBalance metric. It should increase when your function's concurrency is low and decrease when it is high. If it always decreases or does not accumulate enough during low activity to cover peak traffic, you may need to limit your function's concurrency or enable provisioned throughput.

IOPS

Input/output operations per second (IOPS) is a measurement of the number of read and write operations processed by the file system. In general purpose mode, IOPS is limited in favor of lower latency, which is beneficial for most applications.

To monitor and alarm on IOPS in general purpose mode, use the PercentIOLimit metric. If this metric reaches 100%, your function can time out waiting for read and write operations to complete.
Using Lambda with Amazon EventBridge Scheduler

Amazon EventBridge Scheduler is a serverless scheduler that allows you to create, run, and manage tasks from one central, managed service. With EventBridge Scheduler, you can create schedules using cron and rate expressions for recurring patterns, or configure one-time invocations. You can set up flexible time windows for delivery, define retry limits, and set the maximum retention time for unprocessed events.

This page explains how to use EventBridge Scheduler to invoke a Lambda function on a schedule.

Set up the execution role

When you create a new schedule, EventBridge Scheduler must have permission to invoke its target API operation on your behalf. You grant these permissions to EventBridge Scheduler using an execution role. The permission policy you attach to your schedule's execution role defines the required permissions. These permissions depend on the target API you want EventBridge Scheduler to invoke.

When you use the EventBridge Scheduler console to create a schedule, as in the following procedure, EventBridge Scheduler automatically sets up an execution role based on your selected target. If you want to create a schedule using one of the EventBridge Scheduler SDKs, the AWS CLI, or AWS CloudFormation, you must have an existing execution role that grants the permissions EventBridge Scheduler requires to invoke a target. For more information about manually setting up an execution role for your schedule, see Setting up an execution role in the EventBridge Scheduler User Guide.

Create a schedule

To create a schedule by using the console

2. On the Schedules page, choose Create schedule.
3. On the Specify schedule detail page, in the Schedule name and description section, do the following:
   a. For Schedule name, enter a name for your schedule. For example, MyTestSchedule.
b. (Optional) For **Description**, enter a description for your schedule. For example, *My first schedule*.

c. For **Schedule group**, choose a schedule group from the dropdown list. If you don’t have a group, choose **default**. To create a schedule group, choose **create your own schedule**.

You use schedule groups to add tags to groups of schedules.

4. Choose your schedule options.

<table>
<thead>
<tr>
<th>Occurrence</th>
<th>Do this…</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>One-time schedule</strong></td>
<td>For <strong>Date and time</strong>, do the following:</td>
</tr>
<tr>
<td></td>
<td>• Enter a valid date in YYYY/MM/DD format.</td>
</tr>
<tr>
<td></td>
<td>• Enter a timestamp in 24-hour hh:mm format.</td>
</tr>
<tr>
<td></td>
<td>• For <strong>Timezone</strong>, choose the timezone.</td>
</tr>
<tr>
<td><strong>Recurring schedule</strong></td>
<td>a. For <strong>Schedule type</strong>, do one of the following:</td>
</tr>
<tr>
<td></td>
<td>• To use a cron expression to define the schedule, choose <strong>Cron-based schedule</strong> and enter the cron expression.</td>
</tr>
<tr>
<td></td>
<td>• To use a rate expression to define the schedule, choose <strong>Rate-based schedule</strong> and enter the rate expression.</td>
</tr>
<tr>
<td>Occurrence</td>
<td>Do this...</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td>For more information about cron and rate expressions, see <a href="https://docs.aws.amazon.com/eventbridge/latest/userguide/schedule-types.html">Schedule types on EventBridge Scheduler</a> in the Amazon EventBridge Scheduler User Guide.</td>
</tr>
<tr>
<td></td>
<td>b. For <strong>Flexible time window</strong>, choose <strong>Off</strong> to turn off the option, or choose one of the predefined time windows. For example, if you choose <strong>15 minutes</strong> and you set a recurring schedule to invoke its target once every hour, the schedule runs within 15 minutes after the start of every hour.</td>
</tr>
</tbody>
</table>

5. (Optional) If you chose **Recurring schedule** in the previous step, in the **Timeframe** section, do the following:

   a. For **Timezone**, choose a timezone.

   b. For **Start date and time**, enter a valid date in **YYYY/MM/DD** format, and then specify a timestamp in 24-hour **hh:mm** format.

   c. For **End date and time**, enter a valid date in **YYYY/MM/DD** format, and then specify a timestamp in 24-hour **hh:mm** format.

6. Choose **Next**.

7. On the **Select target** page, choose the AWS API operation that EventBridge Scheduler invokes:

   a. Choose **AWS Lambda Invoke**.
b. In the **Invoke** section, select a function or choose **Create new Lambda function**.

c. (Optional) Enter a JSON payload. If you don't enter a payload, EventBridge Scheduler uses an empty event to invoke the function.

8. Choose **Next**.

9. On the **Settings** page, do the following:

   a. To turn on the schedule, under **Schedule state**, toggle **Enable schedule**.

   b. To configure a retry policy for your schedule, under **Retry policy and dead-letter queue (DLQ)**, do the following:

      - Toggle **Retry**.
      - For **Maximum age of event**, enter the maximum **hour(s)** and **min(s)** that EventBridge Scheduler must keep an unprocessed event.
      - The maximum time is 24 hours.
      - For **Maximum retries**, enter the maximum number of times EventBridge Scheduler retries the schedule if the target returns an error.

      The maximum value is 185 retries.

      With retry policies, if a schedule fails to invoke its target, EventBridge Scheduler re-runs the schedule. If configured, you must set the maximum retention time and retries for the schedule.

   c. Choose where EventBridge Scheduler stores undelivered events.

<table>
<thead>
<tr>
<th>Dead-letter queue (DLQ) option</th>
<th>Do this...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Don't store</td>
<td>Choose <strong>None</strong>.</td>
</tr>
<tr>
<td>Store the event in the same AWS account where you're creating the schedule</td>
<td>a. Choose <strong>Select an Amazon SQS queue in my AWS account as a DLQ</strong>.</td>
</tr>
</tbody>
</table>
### Dead-letter queue (DLQ) option

<table>
<thead>
<tr>
<th>Do this...</th>
</tr>
</thead>
<tbody>
<tr>
<td>b. Choose the Amazon Resource Name (ARN) of the Amazon SQS queue.</td>
</tr>
</tbody>
</table>

**Store the event in a different AWS account from where you're creating the schedule**

- **a.** Choose **Specify an Amazon SQS queue in other AWS accounts as a DLQ.**
- **b.** Enter the Amazon Resource Name (ARN) of the Amazon SQS queue.

**d.** To use a customer managed key to encrypt your target input, under **Encryption**, choose **Customize encryption settings (advanced).**

If you choose this option, enter an existing KMS key ARN or choose **Create an AWS KMS key** to navigate to the AWS KMS console. For more information about how EventBridge Scheduler encrypts your data at rest, see **Encryption at rest** in the *Amazon EventBridge Scheduler User Guide.*

**e.** To have EventBridge Scheduler create a new execution role for you, choose **Create new role for this schedule.** Then, enter a name for **Role name.** If you choose this option, EventBridge Scheduler attaches the required permissions necessary for your templated target to the role.

10. **Choose Next.**

11. In the **Review and create schedule** page, review the details of your schedule. In each section, choose **Edit** to go back to that step and edit its details.

12. **Choose Create schedule.**

You can view a list of your new and existing schedules on the **Schedules** page. Under the **Status** column, verify that your new schedule is **Enabled.**

To confirm that EventBridge Scheduler invoked the function, **check the function's Amazon CloudWatch logs.**
Related resources

For more information about EventBridge Scheduler, see the following:

- EventBridge Scheduler User Guide
- EventBridge Scheduler API Reference
- EventBridge Scheduler Pricing
Using AWS Lambda with AWS IoT

AWS IoT provides secure communication between internet-connected devices (such as sensors) and the AWS Cloud. This makes it possible for you to collect, store, and analyze telemetry data from multiple devices.

You can create AWS IoT rules for your devices to interact with AWS services. The AWS IoT Rules Engine provides a SQL-based language to select data from message payloads and send the data to other services, such as Amazon S3, Amazon DynamoDB, and AWS Lambda. You define a rule to invoke a Lambda function when you want to invoke another AWS service or a third-party service.

When an incoming IoT message triggers the rule, AWS IoT invokes your Lambda function asynchronously and passes data from the IoT message to the function.

The following example shows a moisture reading from a greenhouse sensor. The row and pos values identify the location of the sensor. This example event is based on the greenhouse type in the AWS IoT Rules tutorials.

Example AWS IoT message event

```
{
    "row" : "10",
    "pos" : "23",
    "moisture" : "75"
}
```

For asynchronous invocation, Lambda queues the message and retries if your function returns an error. Configure your function with a destination to retain events that your function could not process.

You need to grant permission for the AWS IoT service to invoke your Lambda function. Use the add-permission command to add a permission statement to your function's resource-based policy.

```
aws lambda add-permission --function-name my-function \ 
--statement-id iot-events --action "lambda:InvokeFunction" --principal iot.amazonaws.com
```

You should see the following output:
For more information about how to use Lambda with AWS IoT, see [Creating an AWS Lambda rule](#).
Using AWS Lambda with AWS IoT Events

AWS IoT Events monitors the inputs from multiple IoT sensors and applications to recognize event patterns. Then it takes appropriate actions when events occur. AWS IoT Events receives its inputs as JSON payloads from many sources. AWS IoT Events supports simple events (where each input triggers an event) and complex events (where multiple inputs must occur to trigger the event).

To use AWS IoT Events, you define a detector model, which is a state-machine model of your equipment or process. In addition to states, you define inputs and events for the model. You also define the actions to take when an event occurs. Use a Lambda function for an action when you want to invoke another AWS service (such as Amazon Connect), or take actions in an external application (such as your enterprise resource planning (ERP) application).

When the event occurs, AWS IoT Events invokes your Lambda function asynchronously. It provides information about the detector model and the event that triggered the action. The following example message event is based on the definitions in the AWS IoT Events simple step-by-step example.

Example AWS IoT Events message event

```json
{
    "event": {
        "eventName": "myChargedEvent",
        "eventTime": 1567797571647,
        "payload": {
            "detector": {
                "detectorModelName": "AWS_IoTEvents_Hello_World1567793458261",
                "detectorModelVersion": "4",
                "keyValue": "100009"
            },
            "eventTriggerDetails": {
                "triggerType": "Message",
                "inputName": "AWS_IoTEvents_HelloWorld_VoltageInput",
                "messageId": "64c75a34-068b-4a1d-ae58-c16215dc4efd"
            },
            "actionExecutionId": "49f0f32f-1209-38a7-8a76-d6ca49dd0bc4",
            "state": {
                "variables": {},
                "stateName": "Charged",
                "timers": {}
            }
        }
    }
}
```
The event that is passed into the Lambda function includes the following fields:

- **eventName** – The name for this event in the detector model.
- **eventTime** – The time that the event occurred.
- **detector** – The name and version of the detector model.
- **eventTriggerDetails** – A description of the input that triggered the event.
- **actionExecutionId** – The unique execution identifier of the action.
- **state** – The state of the detector model when the event occurred.
  - **stateName** – The name of the state in the detector model.
  - **timers** – Any timers that are set in this state.
  - **variables** – Any variable values that are set in this state.

You need to grant permission for the AWS IoT Events service to invoke your Lambda function. Use the `aws lambda add-permission` command to add a permission statement to your function's resource-based policy.

```bash
aws lambda add-permission --function-name my-function \
--statement-id iot-events --action "lambda:InvokeFunction" --principal iotevents.amazonaws.com
```

You should see the following output:

```json
{
}
```

For more information about using Lambda with AWS IoT Events, see [Using AWS IoT Events with other services](#)
Using Lambda with self-managed Apache Kafka

### Note

If you want to send data to a target other than a Lambda function or enrich the data before sending it, see [Amazon EventBridge Pipes](https://aws.amazon.com/eventbridge-pipes/).

Lambda supports [Apache Kafka](https://kafka.apache.org) as an **event source**. Apache Kafka is a an open-source event streaming platform that supports workloads such as data pipelines and streaming analytics.

You can use the AWS managed Kafka service Amazon Managed Streaming for Apache Kafka (Amazon MSK), or a self-managed Kafka cluster. For details about using Lambda with Amazon MSK, see [Using Lambda with Amazon MSK](https://docs.aws.amazon.com/lambda/latest/dg/with-amazon-msk.html).

This topic describes how to use Lambda with a self-managed Kafka cluster. In AWS terminology, a self-managed cluster includes non-AWS hosted Kafka clusters. For example, you can host your Kafka cluster with a cloud provider such as [Confluent Cloud](https://confluent.io).

Apache Kafka as an event source operates similarly to using Amazon Simple Queue Service (Amazon SQS) or Amazon Kinesis. Lambda internally polls for new messages from the event source and then synchronously invokes the target Lambda function. Lambda reads the messages in batches and provides these to your function as an event payload. The maximum batch size is configurable. (The default is 100 messages.)

For Kafka-based event sources, Lambda supports processing control parameters, such as batching windows and batch size. For more information, see [Batching behavior](https://docs.aws.amazon.com/lambda/latest/dg/batching.html).

For an example of how to use self-managed Kafka as an event source, see [Using self-hosted Apache Kafka as an event source for AWS Lambda](https://aws.amazon.com/blogs/compute/) on the AWS Compute Blog.

**Topics**

- Example event
- Kafka cluster authentication
- Managing API access and permissions
- Authentication and authorization errors
- Network configuration
Adding a Kafka cluster as an event source

Using a Kafka cluster as an event source

Polling and stream starting positions

Auto scaling of the Kafka event source

Event source API operations

Event source mapping errors

Amazon CloudWatch metrics

Self-managed Apache Kafka configuration parameters

Example event

Lambda sends the batch of messages in the event parameter when it invokes your Lambda function. The event payload contains an array of messages. Each array item contains details of the Kafka topic and Kafka partition identifier, together with a timestamp and a base64-encoded message.

```json
{
    "eventSource": "SelfManagedKafka",
    "bootstrapServers": "b-2.demo-cluster-1.a1bcde.c1.kafka.us-east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-east-1.amazonaws.com:9092",
    "records": {
        "mytopic-0": [
            {
                "topic": "mytopic",
                "partition": 0,
                "offset": 15,
                "timestamp": 1545084650987,
                "timestampType": "CREATE_TIME",
                "key": "abcDEFGhiJKLmnoPQstuVWXyz1234==",
                "value": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "headers": [
                    {
                        "headerKey": [104, 101, 97, 100, 101]}
```
Kafka cluster authentication

Lambda supports several methods to authenticate with your self-managed Apache Kafka cluster. Make sure that you configure the Kafka cluster to use one of these supported authentication methods. For more information about Kafka security, see the Security section of the Kafka documentation.

VPC access

If only Kafka users within your VPC access your Kafka brokers, you must configure the Kafka event source for Amazon Virtual Private Cloud (Amazon VPC) access.

SASL/SCRAM authentication

Lambda supports Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) authentication with Transport Layer Security (TLS) encryption (SASL_SSL). Lambda sends the encrypted credentials to authenticate with the cluster. Lambda doesn't support SASL/SCRAM with plaintext (SASL_PLAINTEXT). For more information about SASL/SCRAM authentication, see RFC 5802.

Lambda also supports SASL/PLAIN authentication. Because this mechanism uses clear text credentials, the connection to the server must use TLS encryption to ensure that the credentials are protected.

For SASL authentication, you store the sign-in credentials as a secret in AWS Secrets Manager. For more information about using Secrets Manager, see Tutorial: Create and retrieve a secret in the AWS Secrets Manager User Guide.
Important

To use Secrets Manager for authentication, secrets must be stored in the same AWS region as your Lambda function.

Mutual TLS authentication

Mutual TLS (mTLS) provides two-way authentication between the client and server. The client sends a certificate to the server for the server to verify the client, and the server sends a certificate to the client for the client to verify the server.

In self-managed Apache Kafka, Lambda acts as the client. You configure a client certificate (as a secret in Secrets Manager) to authenticate Lambda with your Kafka brokers. The client certificate must be signed by a CA in the server's trust store.

The Kafka cluster sends a server certificate to Lambda to authenticate the Kafka brokers with Lambda. The server certificate can be a public CA certificate or a private CA/self-signed certificate. The public CA certificate must be signed by a certificate authority (CA) that's in the Lambda trust store. For a private CA/self-signed certificate, you configure the server root CA certificate (as a secret in Secrets Manager). Lambda uses the root certificate to verify the Kafka brokers.

For more information about mTLS, see [Introducing mutual TLS authentication for Amazon MSK as an event source](#).

Configuring the client certificate secret

The CLIENT_CERTIFICATE_TLS_AUTH secret requires a certificate field and a private key field. For an encrypted private key, the secret requires a private key password. Both the certificate and private key must be in PEM format.

Note

Lambda supports the PBES1 (but not PBES2) private key encryption algorithms.

The certificate field must contain a list of certificates, beginning with the client certificate, followed by any intermediate certificates, and ending with the root certificate. Each certificate must start on a new line with the following structure:
Secrets Manager supports secrets up to 65,536 bytes, which is enough space for long certificate chains.

The private key must be in [PKCS #8](https://www.ietf.org/rfc/rfc2314.txt) format, with the following structure:

```
-----BEGIN PRIVATE KEY-----
<private key contents>
-----END PRIVATE KEY-----
```

For an encrypted private key, use the following structure:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
<private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

The following example shows the contents of a secret for mTLS authentication using an encrypted private key. For an encrypted private key, include the private key password in the secret.

```
{"privateKeyPassword":"testpassword",
 "certificate":"-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRiytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
... 
CMV0QyE2KmI36dg4lMzNjAEBZiCORpIMo40s1cRqtFHXoa1QQdI1xk
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNz6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
... 
Q0iowbbk5wXCheYSANQIftZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3P5oAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
 "privateKey":"-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzVBgkqzhki1G9w0BBQ0wSDAnBgkqzhkiG9w0BBQwGiuAFCaK5hT/X7Kjmgp
... 
QrSekqF+kWzmB6nAFszg01nAoAytLtvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+impl0azDz+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
Configuring the server root CA certificate secret

You create this secret if your Kafka brokers use TLS encryption with certificates signed by a private CA. You can use TLS encryption for VPC, SASL/SCRAM, SASL/PLAIN, or mTLS authentication.

The server root CA certificate secret requires a field that contains the Kafka broker's root CA certificate in PEM format. The following example shows the structure of the secret.

```
{"certificate":"-----BEGIN CERTIFICATE-----
MIID7zCCAtgAwIBAgIBADANBgkqhkiG9w0BAQsFADCBmDELMAkGA1UEBhMCVVMx
EDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxJTAjBgNVBAoT
HFN0YXJmaWVsZCBUZWNobm9sb2dpZXMsIEluYy4xOzA5BgNVBAMTMlN0YXJmaWVs
ZCBTZVJ2aWNlcyBSb290IENlcnRpZmljYXRlIEF1dG...
-----END CERTIFICATE-----"}
```

Managing API access and permissions

In addition to accessing your self-managed Kafka cluster, your Lambda function needs permissions to perform various API actions. You add these permissions to the function's execution role. If your users need access to any API actions, add the required permissions to the identity policy for the AWS Identity and Access Management (IAM) user or role.

**Required Lambda function permissions**

To create and store logs in a log group in Amazon CloudWatch Logs, your Lambda function must have the following permissions in its execution role:

- `logs:CreateLogGroup`
- `logs:CreateLogStream`
- `logs:PutLogEvents`

**Optional Lambda function permissions**

Your Lambda function might also need permissions to:

- Describe your Secrets Manager secret.
• Access your AWS Key Management Service (AWS KMS) customer managed key.
• Access your Amazon VPC.
• Send records of failed invocations to a destination.

Secrets Manager and AWS KMS permissions

Depending on the type of access control that you're configuring for your Kafka brokers, your Lambda function might need permission to access your Secrets Manager secret or to decrypt your AWS KMS customer managed key. To access these resources, your function's execution role must have the following permissions:

• `secretsmanager:GetSecretValue`
• `kms:Decrypt`

VPC permissions

If only users within a VPC can access your self-managed Apache Kafka cluster, your Lambda function must have permission to access your Amazon VPC resources. These resources include your VPC, subnets, security groups, and network interfaces. To access these resources, your function's execution role must have the following permissions:

• `ec2:CreateNetworkInterface`
• `ec2:DescribeNetworkInterfaces`
• `ec2:DescribeVpcs`
• `ec2:DeleteNetworkInterface`
• `ec2:DescribeSubnets`
• `ec2:DescribeSecurityGroups`

Sending records to a destination

If you want to send records of failed invocations to an on-failure destination, your Lambda function must have permission to send these records. For Kafka event source mappings, you can choose between an Amazon SNS topic, Amazon SQS queue, or Amazon S3 bucket as the destination. To send records to an SNS topic, your function's execution role must have the following permission:

• `sns:Publish`
To send records to an SQS queue, your function's execution role must have the following permission:

- sqs:SendMessage

To send records to an S3 bucket, your function's execution role must have the following permissions:

- s3:PutObject
- s3:ListBuckets

Additionally, if you configured a KMS key on your destination, Lambda needs the following permissions depending on the destination type:

- If you've enabled encryption with your own KMS key for an S3 destination, kms:GenerateDataKey is required. If the KMS key and S3 bucket destination are in a different account from your Lambda function and execution role, configure the KMS key to trust the execution role to allow kms:GenerateDataKey.

- If you've enabled encryption with your own KMS key for SQS destination, kms:Decrypt and kms:GenerateDataKey are required. If the KMS key and SQS queue destination are in a different account from your Lambda function and execution role, configure the KMS key to trust the execution role to allow kms:Decrypt, kms:GenerateDataKey, kms:DescribeKey, and kms:ReEncrypt.

- If you've enabled encryption with your own KMS key for SNS destination, kms:Decrypt and kms:GenerateDataKey are required. If the KMS key and SNS topic destination are in a different account from your Lambda function and execution role, configure the KMS key to trust the execution role to allow kms:Decrypt, kms:GenerateDataKey, kms:DescribeKey, and kms:ReEncrypt.

### Adding permissions to your execution role

To access other AWS services that your self-managed Apache Kafka cluster uses, Lambda uses the permissions policies that you define in your Lambda function's execution role.

By default, Lambda is not permitted to perform the required or optional actions for a self-managed Apache Kafka cluster. You must create and define these actions in an IAM trust policy,
and then attach the policy to your execution role. This example shows how you might create a policy that allows Lambda to access your Amazon VPC resources.

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Resource":"*"
      }
   ]
}
```

For information about creating a JSON policy document in the IAM console, see Creating policies on the JSON tab in the IAM User Guide.

**Granting users access with an IAM policy**

By default, users and roles don't have permission to perform event source API operations. To grant access to users in your organization or account, you create or update an identity-based policy. For more information, see Controlling access to AWS resources using policies in the IAM User Guide.

**Authentication and authorization errors**

If any of the permissions required to consume data from the Kafka cluster are missing, Lambda displays one of the following error messages in the event source mapping under LastProcessingResult.

**Error messages**

- Cluster failed to authorize Lambda
- SASL authentication failed
- Server failed to authenticate Lambda
• **Lambda failed to authenticate server**

• **Provided certificate or private key is invalid**

**Cluster failed to authorize Lambda**

For SASL/SCRAM or mTLS, this error indicates that the provided user doesn't have all of the following required Kafka access control list (ACL) permissions:

• DescribeConfigs Cluster

• Describe Group

• Read Group

• Describe Topic

• Read Topic

When you create Kafka ACLs with the required `kafka-cluster` permissions, specify the topic and group as resources. The topic name must match the topic in the event source mapping. The group name must match the event source mapping's UUID.

After you add the required permissions to the execution role, it might take several minutes for the changes to take effect.

**SASL authentication failed**

For SASL/SCRAM or SASL/PLAIN, this error indicates that the provided sign-in credentials aren't valid.

**Server failed to authenticate Lambda**

This error indicates that the Kafka broker failed to authenticate Lambda. This can occur for any of the following reasons:

• You didn't provide a client certificate for mTLS authentication.

• You provided a client certificate, but the Kafka brokers aren't configured to use mTLS authentication.

• A client certificate isn't trusted by the Kafka brokers.
Lambda failed to authenticate server

This error indicates that Lambda failed to authenticate the Kafka broker. This can occur for any of the following reasons:

- The Kafka brokers use self-signed certificates or a private CA, but didn't provide the server root CA certificate.
- The server root CA certificate doesn't match the root CA that signed the broker's certificate.
- Hostname validation failed because the broker's certificate doesn't contain the broker's DNS name or IP address as a subject alternative name.

Provided certificate or private key is invalid

This error indicates that the Kafka consumer couldn't use the provided certificate or private key. Make sure that the certificate and key use PEM format, and that the private key encryption uses a PBES1 algorithm.

Network configuration

If you configure Amazon VPC access to your Kafka brokers, Lambda must have access to the Amazon VPC resources your Kafka cluster resides in. We recommend that you deploy AWS PrivateLink VPC endpoints for Lambda and AWS Security Token Service (AWS STS). If the broker uses authentication, also deploy a VPC endpoint for Secrets Manager. If you configured an on-failure destination, also deploy a VPC endpoint for the destination service.

Alternatively, ensure that the VPC associated with your Kafka cluster includes one NAT gateway per public subnet. For more information, see Internet and service access for VPC-connected functions.

If you use VPC endpoints, you must also configure them to enable private DNS names.

Configure your Amazon VPC security groups with the following rules (at minimum):

- Inbound rules – Allow all traffic on the Kafka broker port for the security groups specified for your event source. Kafka uses port 9092 by default.
- Outbound rules – Allow all traffic on port 443 for all destinations. Allow all traffic on the Kafka broker port for the security groups specified for your event source. Kafka uses port 9092 by default.
• If you are using VPC endpoints instead of a NAT gateway, the security groups associated with the VPC endpoints must allow all inbound traffic on port 443 from the event source's security groups.

For more information about configuring the network, see Setting up AWS Lambda with an Apache Kafka cluster within a VPC on the AWS Compute Blog.

**Adding a Kafka cluster as an event source**

To create an event source mapping, add your Kafka cluster as a Lambda function trigger using the Lambda console, an AWS SDK, or the AWS Command Line Interface (AWS CLI).

This section describes how to create an event source mapping using the Lambda console and the AWS CLI.

⚠️ **Note**

When you update, disable, or delete an event source mapping for self-managed Apache Kafka, it can take up to 15 minutes for your changes to take effect. Before this period has elapsed, your event source mapping may continue to process events and invoke your function using your previous settings. This is true even when the status of the event source mapping displayed in the console indicates that your changes have been applied.

**Prerequisites**

• A self-managed Apache Kafka cluster. Lambda supports Apache Kafka version 0.10.1.0 and later.

• An execution role with permission to access the AWS resources that your self-managed Kafka cluster uses.

**Customizable consumer group ID**

When setting up Kafka as an event source, you can specify a consumer group ID. This consumer group ID is an existing identifier for the Kafka consumer group that you want your Lambda function to join. You can use this feature to seamlessly migrate any ongoing Kafka record processing setups from other consumers to Lambda.
If you specify a consumer group ID and there are other active pollers within that consumer group, Kafka distributes messages across all consumers. In other words, Lambda doesn't receive all message for the Kafka topic. If you want Lambda to handle all messages in the topic, turn off any other pollers in that consumer group.

Additionally, if you specify a consumer group ID, and Kafka finds a valid existing consumer group with the same ID, Lambda ignores the StartingPosition parameter for your event source mapping. Instead, Lambda begins processing records according to the committed offset of the consumer group. If you specify a consumer group ID, and Kafka cannot find an existing consumer group, then Lambda configures your event source with the specified StartingPosition.

The consumer group ID that you specify must be unique among all your Kafka event sources. After creating a Kafka event source mapping with the consumer group ID specified, you cannot update this value.

**On-failure destinations**

To retain records of failed invocations or oversized payloads from your Kafka event source, configure an on-failure destination to your function. When an invocation fails, Lambda sends a JSON record containing details of the invocation to your destination.

You can choose between an Amazon SNS topic, Amazon SQS queue, or Amazon S3 bucket as your destination. For SNS topic or SQS queue destinations, Lambda sends the record metadata to the destination. For S3 bucket destinations, Lambda sends the entire invocation record along with the metadata to the destination.

For Lambda to successfully send records to your chosen destination, ensure that your function's execution role contains the relevant permissions. The table also describes how each destination type receives the JSON invocation record.

<table>
<thead>
<tr>
<th>Destination type</th>
<th>Supported for the following event sources</th>
<th>Required permissions</th>
<th>Destination-specific JSON format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon SQS queue</td>
<td>• Kinesis</td>
<td>• sqs:SendMessage</td>
<td>Lambda passes the invocation record metadata as the</td>
</tr>
</tbody>
</table>
### Destination type

<table>
<thead>
<tr>
<th>Supported for the following event sources</th>
<th>Required permissions</th>
<th>Destination-specific JSON format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Managed Apache Kafka</td>
<td></td>
<td>Message to the destination.</td>
</tr>
<tr>
<td>Amazon SNS topic</td>
<td></td>
<td>Lambda passes the invocation record metadata as the Message to the destination.</td>
</tr>
<tr>
<td>Amazon S3 bucket</td>
<td></td>
<td>Lambda stores the invocation record along with its metadata at the destination.</td>
</tr>
</tbody>
</table>

#### Amazon SNS topic
- Kinesis
- DynamoDB
- Self-managed Apache Kafka and Managed Apache Kafka
  - `sns:Publish`

#### Amazon S3 bucket
- Self-managed Apache Kafka and Managed Apache Kafka
  - `s3:PutObject`
  - `s3:ListBuckets`

### Tip
As a best practice, include the minimum permissions required only in your execution role.

### SNS and SQS destinations
The following example shows what Lambda sends to an SNS topic or SQS queue destination for a failed Kafka event source invocation. Each of the keys under `recordsInfo` contains both the Kafka topic and partition, separated by a hyphen. For example, for the key "Topic-0", Topic is the Kafka topic, and 0 is the partition. For each topic and partition, you can use the offsets and timestamp data to find the original invocation records.

```json
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
```
S3 destinations

For S3 destinations, Lambda sends the entire invocation record along with the metadata to the destination. The following example shows that Lambda sends to an S3 bucket destination for a failed Kafka event source invocation. In addition to all of the fields from the previous example for SQS and SNS destinations, the payload field contains the original invocation record as an escaped JSON string.
Adding a Kafka cluster as an event source

```json
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset": "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset": "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z"
            },
            "Topic-1": {
                "firstRecordOffset": "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset": "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z"
            }
        }
    },
    "payload": "<Whole Event>" // Only available in S3
}
Tip
We recommend enabling S3 versioning on your destination bucket.

Configuring on-failure destinations

To configure an on-failure destination using the console, follow these steps:

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Under Function overview, choose Add destination.
4. For Source, choose Event source mapping invocation.
5. For Event source mapping, choose an event source that's configured for this function.
6. For Condition, select On failure. For event source mapping invocations, this is the only accepted condition.
7. For Destination type, choose the destination type that Lambda sends invocation records to.
8. For Destination, choose a resource.
9. Choose Save.

You can also configure an on-failure destination using the Lambda API. For example, the following CreateEventSourceMapping CLI command adds an SQS on-failure destination to MyFunction:

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-east-1:123456789012:dest-queue"}}'
```

The following UpdateEventSourceMapping CLI command adds an S3 on-failure destination to the Kafka event source associated with the input uuid:

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

To remove a destination, supply an empty string as the argument to the destination-config parameter:
Adding a self-managed Kafka cluster (console)

Follow these steps to add your self-managed Apache Kafka cluster and a Kafka topic as a trigger for your Lambda function.

**To add an Apache Kafka trigger to your Lambda function (console)**

1. Open the Functions page of the Lambda console.
2. Choose the name of your Lambda function.
4. Under Trigger configuration, do the following:

   a. Choose the Apache Kafka trigger type.

   b. For Bootstrap servers, enter the host and port pair address of a Kafka broker in your cluster, and then choose Add. Repeat for each Kafka broker in the cluster.

   c. For Topic name, enter the name of the Kafka topic used to store records in the cluster.

   d. (Optional) For Batch size, enter the maximum number of records to receive in a single batch.

   e. For Batch window, enter the maximum amount of seconds that Lambda spends gathering records before invoking the function.

   f. (Optional) For Consumer group ID, enter the ID of a Kafka consumer group to join.

   g. (Optional) For Starting position, choose Latest to start reading the stream from the latest record, Trim horizon to start at the earliest available record, or At timestamp to specify a timestamp to start reading from.

   h. (Optional) For VPC, choose the Amazon VPC for your Kafka cluster. Then, choose the VPC subnets and VPC security groups.

      This setting is required if only users within your VPC access your brokers.

   i. (Optional) For Authentication, choose Add, and then do the following:

      i. Choose the access or authentication protocol of the Kafka brokers in your cluster.
If your Kafka broker uses SASL/PLAIN authentication, choose **BASIC_AUTH**.

If your broker uses SASL/SCRAM authentication, choose one of the **SASL_SCRAM** protocols.

If you're configuring mTLS authentication, choose the **CLIENT_CERTIFICATE_TLS_AUTH** protocol.

ii. For SASL/SCRAM or mTLS authentication, choose the Secrets Manager secret key that contains the credentials for your Kafka cluster.

j. (Optional) For **Encryption**, choose the Secrets Manager secret containing the root CA certificate that your Kafka brokers use for TLS encryption, if your Kafka brokers use certificates signed by a private CA.

This setting applies to TLS encryption for SASL/SCRAM or SASL/PLAIN, and to mTLS authentication.

k. To create the trigger in a disabled state for testing (recommended), clear **Enable trigger**. Or, to enable the trigger immediately, select **Enable trigger**.

5. To create the trigger, choose **Add**.

### Adding a self-managed Kafka cluster (AWS CLI)

Use the following example AWS CLI commands to create and view a self-managed Apache Kafka trigger for your Lambda function.

#### Using SASL/SCRAM

If Kafka users access your Kafka brokers over the internet, specify the Secrets Manager secret that you created for SASL/SCRAM authentication. The following example uses the `create-event-source-mapping` AWS CLI command to map a Lambda function named `my-kafka-function` to a Kafka topic named `AWSKafkaTopic`.

```bash
aws lambda create-event-source-mapping --topics AWSKafkaTopic
   --source-access-configuration
   Type=SASL_SCRAM_512_AUTH,URI=arn:aws:secretsmanager:us-east-1:01234567890:secret:MyBrokerSecretName
```
Using a VPC

If only Kafka users within your VPC access your Kafka brokers, you must specify your VPC, subnets, and VPC security group. The following example uses the `create-event-source-mapping` AWS CLI command to map a Lambda function named `my-kafka-function` to a Kafka topic named `AWSKafkaTopic`.

```bash
aws lambda create-event-source-mapping
  --topics AWSKafkaTopic
  --source-access-configuration '[{"Type": "VPC_SUBNET", "URI": "subnet:subnet-0011001100"},
                                 {{"Type": "VPC_SUBNET", "URI": "subnet:subnet-0022002200"},
                                 {{"Type": "VPC_SECURITY_GROUP", "URI": "security_group:sg-0123456789"}}]
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS": ["abc3.xyz.com:9092",
                             "abc2.xyz.com:9092"]}}'
```

Viewing the status using the AWS CLI

The following example uses the `get-event-source-mapping` AWS CLI command to describe the status of the event source mapping that you created.

```bash
aws lambda get-event-source-mapping
  --uuid dh38738e-992b-343a-1077-3478934hjkfd7
```

Using a Kafka cluster as an event source

When you add your Apache Kafka cluster as a trigger for your Lambda function, the cluster is used as an event source.

Lambda reads event data from the Kafka topics that you specify as Topics in a `CreateEventSourceMapping` request, based on the StartingPosition that you specify. After successful processing, your Kafka topic is committed to your Kafka cluster.

If you specify the StartingPosition as LATEST, Lambda starts reading from the latest message in each partition belonging to the topic. Because there can be some delay after trigger
configuration before Lambda starts reading the messages, Lambda doesn't read any messages produced during this window.

Lambda processes records from one or more Kafka topic partitions that you specify and sends a JSON payload to your function. When more records are available, Lambda continues processing records in batches, based on the BatchSize value that you specify in a CreateEventSourceMapping request, until your function catches up with the topic.

If your function returns an error for any of the messages in a batch, Lambda retries the whole batch of messages until processing succeeds or the messages expire. You can send records that fail all retry attempts to an on-failure destination for later processing.

⚠️ Note

While Lambda functions typically have a maximum timeout limit of 15 minutes, event source mappings for Amazon MSK, self-managed Apache Kafka, Amazon DocumentDB, and Amazon MQ for ActiveMQ and RabbitMQ only support functions with maximum timeout limits of 14 minutes. This constraint ensures that the event source mapping can properly handle function errors and retries.

Polling and stream starting positions

Be aware that stream polling during event source mapping creation and updates is eventually consistent.

- During event source mapping creation, it may take several minutes to start polling events from the stream.
- During event source mapping updates, it may take several minutes to stop and restart polling events from the stream.

This behavior means that if you specify LATEST as the starting position for the stream, the event source mapping could miss events during creation or updates. To ensure that no events are missed, specify the stream starting position as TRIM_HORIZON or AT_TIMESTAMP.
Auto scaling of the Kafka event source

When you initially create an Apache Kafka event source, Lambda allocates one consumer to process all partitions in the Kafka topic. Each consumer has multiple processors running in parallel to handle increased workloads. Additionally, Lambda automatically scales up or down the number of consumers, based on workload. To preserve message ordering in each partition, the maximum number of consumers is one consumer per partition in the topic.

In one-minute intervals, Lambda evaluates the consumer offset lag of all the partitions in the topic. If the lag is too high, the partition is receiving messages faster than Lambda can process them. If necessary, Lambda adds or removes consumers from the topic. The scaling process of adding or removing consumers occurs within three minutes of evaluation.

If your target Lambda function is overloaded, Lambda reduces the number of consumers. This action reduces the workload on the function by reducing the number of messages that consumers can retrieve and send to the function.

To monitor the throughput of your Kafka topic, you can view the Apache Kafka consumer metrics, such as consumer_lag and consumer_offset. To check how many function invocations occur in parallel, you can also monitor the concurrency metrics for your function.

Event source API operations

When you add your Kafka cluster as an event source for your Lambda function using the Lambda console, an AWS SDK, or the AWS CLI, Lambda uses APIs to process your request.

To manage an event source with the AWS Command Line Interface (AWS CLI) or an AWS SDK, you can use the following API operations:

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping
Event source mapping errors

When you add your Apache Kafka cluster as an event source for your Lambda function, if your function encounters an error, your Kafka consumer stops processing records. Consumers of a topic partition are those that subscribe to, read, and process your records. Your other Kafka consumers can continue processing records, provided they don't encounter the same error.

To determine the cause of a stopped consumer, check the StateTransitionReason field in the response of EventSourceMapping. The following list describes the event source errors that you can receive:

**ESM_CONFIG_NOT_VALID**

The event source mapping configuration isn't valid.

**EVENT_SOURCE_AUTHN_ERROR**

Lambda couldn't authenticate the event source.

**EVENT_SOURCE_AUTHZ_ERROR**

Lambda doesn't have the required permissions to access the event source.

**FUNCTION_CONFIG_NOT_VALID**

The function configuration isn't valid.

Note

If your Lambda event records exceed the allowed size limit of 6 MB, they can go unprocessed.

Amazon CloudWatch metrics

Lambda emits the OffsetLag metric while your function processes records. The value of this metric is the difference in offset between the last record written to the Kafka event source topic and the last record that your function's consumer group processed. You can use OffsetLag to estimate the latency between when a record is added and when your consumer group processes it.

An increasing trend in OffsetLag can indicate issues with pollers in your function's consumer group. For more information, see Working with Lambda function metrics.
Self-managed Apache Kafka configuration parameters

All Lambda event source types share the same `CreateEventSourceMapping` and `UpdateEventSourceMapping` API operations. However, only some of the parameters apply to Apache Kafka.

**Event source parameters that apply to self-managed Apache Kafka**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Default</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BatchSize</td>
<td>N</td>
<td>100</td>
<td>Maximum: 10,000</td>
</tr>
<tr>
<td>Enabled</td>
<td>N</td>
<td>Enabled</td>
<td></td>
</tr>
<tr>
<td>FunctionName</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FilterCriteria</td>
<td>N</td>
<td></td>
<td>Lambda event filtering</td>
</tr>
<tr>
<td>MaximumBatchingWindowInSeconds</td>
<td>N</td>
<td>500 ms</td>
<td>Batching behavior</td>
</tr>
<tr>
<td>SelfManagedEventSource</td>
<td>Y</td>
<td></td>
<td>List of Kafka Brokers. Can set only on Create</td>
</tr>
<tr>
<td>SelfManagedKafkaEventSourceConfig</td>
<td>N</td>
<td></td>
<td>Contains the ConsumerGroupId field which defaults to a unique value. Can set only on Create</td>
</tr>
<tr>
<td>SourceAccessConfigurations</td>
<td>N</td>
<td>No credentials</td>
<td>VPC information or authentication credentials for the cluster For SASL_PLAIN, set to BASIC_AUTH</td>
</tr>
<tr>
<td>Parameter</td>
<td>Required</td>
<td>Default</td>
<td>Notes</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
<td>---------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>StartingPosition</td>
<td>Y</td>
<td></td>
<td>AT_TIMESTAMP, TRIM_HORIZON, or LATEST</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Can set only on Create</td>
</tr>
<tr>
<td>StartingPositionTime</td>
<td>N</td>
<td></td>
<td>Required if StartingPosition is set to AT_TIMESTAMP</td>
</tr>
<tr>
<td>Topics</td>
<td>Y</td>
<td></td>
<td>Topic name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Can set only on Create</td>
</tr>
</tbody>
</table>
Using AWS Lambda with Amazon Kinesis Data Firehose

Amazon Kinesis Data Firehose captures, transforms, and loads streaming data into downstream services such as Managed Service for Apache Flink or Amazon S3. You can write Lambda functions to request additional, customized processing of the data before it is sent downstream.

Example Amazon Kinesis Data Firehose message event

```
{
    "invocationId": "invoked123",
    "deliveryStreamArn": "aws:lambda:events",
    "region": "us-west-2",
    "records": [
        {
            "data": "SGVsbG8gV29ybGQ=",
            "recordId": "record1",
            "approximateArrivalTimestamp": 1510772160000,
            "kinesisRecordMetadata": {
                "shardId": "shardId-000000000000",
                "partitionKey": "4d1ad2b9-24f8-4b9d-a088-76e9947c317a",
                "approximateArrivalTimestamp": "2012-04-23T18:25:43.511Z",
                "sequenceNumber": "49546986683135544286507457936321625675700192471156785154",
                "subsequenceNumber": ""
            }
        },
        {
            "data": "SGVsbG8gV29ybGQ=",
            "recordId": "record2",
            "approximateArrivalTimestamp": 151077216000,
            "kinesisRecordMetadata": {
                "shardId": "shardId-000000000001",
                "partitionKey": "4d1ad2b9-24f8-4b9d-a088-76e9947c318a",
                "approximateArrivalTimestamp": "2012-04-23T19:25:43.511Z",
                "sequenceNumber": "49546986683135544286507457936321625675700192471156785155",
                "subsequenceNumber": ""
            }
        }
    ]
}
```

For more information, see [Amazon Kinesis Data Firehose data transformation](https://docs.aws.amazon.com/kinesis/latest/dev/amazon-kinesis-data-firehose-data-transformation.html) in the Kinesis Data Firehose Developer Guide.
Using AWS Lambda with Amazon Kinesis

Note
If you want to send data to a target other than a Lambda function or enrich the data before sending it, see Amazon EventBridge Pipes.

You can use an AWS Lambda function to process records in an Amazon Kinesis data stream.

A Kinesis data stream is a set of shards. Each shard contains a sequence of data records. A consumer is an application that processes the data from a Kinesis data stream. You can map a Lambda function to a shared-throughput consumer (standard iterator), or to a dedicated-throughput consumer with enhanced fan-out.

For standard iterators, Lambda polls each shard in your Kinesis stream for records using HTTP protocol. The event source mapping shares read throughput with other consumers of the shard.

To minimize latency and maximize read throughput, you can create a data stream consumer with enhanced fan-out. Stream consumers get a dedicated connection to each shard that doesn't impact other applications reading from the stream. The dedicated throughput can help if you have many applications reading the same data, or if you're reprocessing a stream with large records. Kinesis pushes records to Lambda over HTTP/2.

For details about Kinesis data streams, see Reading Data from Amazon Kinesis Data Streams.

Sections

- Example event
- Polling and batching streams
- Polling and stream starting position
- Configuring your data stream and function
- Execution role permissions
- Configuring a stream as an event source
- Filtering Kinesis events
- Event source mapping API
- Error handling
- Amazon CloudWatch metrics
Example event

Example

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID": "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"
        },
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber": "49590338271490256608559692540925702759324208523137515618",
                "data": "VGhpcyBpcyBvbmx5IGEgdGVzdC4=",
                "approximateArrivalTimestamp": 1545084711.166
            }
        }
    ]
}
```
Polling and batching streams

Lambda reads records from the data stream and invokes your function synchronously with an event that contains stream records. Lambda reads records in batches and invokes your function to process records from the batch. Each batch contains records from a single shard/data stream.

By default, Lambda invokes your function as soon as records are available. If the batch that Lambda reads from the event source has only one record in it, Lambda sends only one record to the function. To avoid invoking the function with a small number of records, you can tell the event source to buffer records for up to 5 minutes by configuring a batching window. Before invoking the function, Lambda continues to read records from the event source until it has gathered a full batch, the batching window expires, or the batch reaches the payload limit of 6 MB. For more information, see Batching behavior.

If your function returns an error, Lambda retries the batch until processing succeeds or the data expires. To avoid stalled shards, you can configure the event source mapping to retry with a smaller batch size, limit the number of retries, or discard records that are too old. To retain discarded events, you can configure the event source mapping to send details about failed batches to a standard SQS queue or standard SNS topic.

To increase concurrency, you can also process multiple batches from each shard in parallel. Lambda can process up to 10 batches in each shard simultaneously. If you increase the number of concurrent batches per shard, Lambda still ensures in-order processing at the partition-key level.

Configure the ParallelizationFactor setting to process one shard of a Kinesis or DynamoDB data stream with more than one Lambda invocation simultaneously. You can specify the number of concurrent batches that Lambda polls from a shard via a parallelization factor from 1 (default)
to 10. For example, when you set ParallelizationFactor to 2, you can have 200 concurrent Lambda invocations at maximum to process 100 Kinesis data shards (though in practice, you may see a different values for the ConcurrentExecutions metric). This helps scale up the processing throughput when the data volume is volatile and the IteratorAge is high. Note that parallelization factor will not work if you are using Kinesis aggregation. For more information, see New AWS Lambda scaling controls for Kinesis and DynamoDB event sources. Also, see the Serverless Data Processing on AWS workshop for complete tutorials.

Polling and stream starting position

Be aware that stream polling during event source mapping creation and updates is eventually consistent.

- During event source mapping creation, it may take several minutes to start polling events from the stream.
- During event source mapping updates, it may take several minutes to stop and restart polling events from the stream.

This behavior means that if you specify LATEST as the starting position for the stream, the event source mapping could miss events during creation or updates. To ensure that no events are missed, specify the stream starting position as TRIM_HORIZON or AT_TIMESTAMP.

Configuring your data stream and function

Your Lambda function is a consumer application for your data stream. It processes one batch of records at a time from each shard. You can map a Lambda function to a data stream (standard iterator), or to a consumer of a stream (enhanced fan-out).

For standard iterators, Lambda polls each shard in your Kinesis stream for records at a base rate of once per second. When more records are available, Lambda keeps processing batches until the function catches up with the stream. The event source mapping shares read throughput with other consumers of the shard.

To minimize latency and maximize read throughput, create a data stream consumer with enhanced fan-out. Enhanced fan-out consumers get a dedicated connection to each shard that doesn't impact other applications reading from the stream. Stream consumers use HTTP/2 to reduce latency by pushing records to Lambda over a long-lived connection and by compressing request headers. You can create a stream consumer with the Kinesis RegisterStreamConsumer API.
You should see the following output:

```
{
    "Consumer": {
        "ConsumerName": "con1",
        "ConsumerStatus": "CREATING",
        "ConsumerCreationTimestamp": 1540591608.0
    }
}
```

To increase the speed at which your function processes records, add shards to your data stream. Lambda processes records in each shard in order. It stops processing additional records in a shard if your function returns an error. With more shards, there are more batches being processed at once, which lowers the impact of errors on concurrency.

If your function can't scale up to handle the total number of concurrent batches, request a quota increase or reserve concurrency for your function.

### Execution role permissions

Lambda needs the following permissions to manage resources that are related to your Kinesis data stream. Add them to your function's `execution role`.

- `kinesis:DescribeStream`
- `kinesis:DescribeStreamSummary`
- `kinesis:GetRecords`
- `kinesis:GetShardIterator`
- `kinesis:ListShards`
- `kinesis:ListStreams`
- `kinesis:SubscribeToShard`

The AWSLambdaKinesisExecutionRole managed policy includes these permissions. For more information, see Lambda execution role.
If your Kinesis data stream and Lambda function are in different accounts, ensure that your stream resource grants \texttt{kinesis:DescribeStream} permissions to your Lambda function's execution role or account.

Additionally, when creating your event source mapping from the console, you must have the \texttt{kinesis:ListStreams} and \texttt{kinesis:ListStreamConsumers} permissions.

To send records of failed batches to a standard SQS queue or standard SNS topic, your function needs additional permissions. Each destination service requires a different permission, as follows:

- **Amazon SQS** – \texttt{sqs:SendMessage}
- **Amazon SNS** – \texttt{sns:Publish}

**Configuring a stream as an event source**

Create an event source mapping to tell Lambda to send records from your data stream to a Lambda function. You can create multiple event source mappings to process the same data with multiple Lambda functions, or to process items from multiple data streams with a single function. When processing items from multiple data streams, each batch will only contain records from a single shard/stream.

To configure your function to read from Kinesis in the Lambda console, create a **Kinesis** trigger.

**To create a trigger**

1. Open the **Functions page** of the Lambda console.
2. Choose the name of a function.
3. Under **Function overview**, choose **Add trigger**.
4. Choose a trigger type.
5. Configure the required options, and then choose **Add**.

Lambda supports the following options for Kinesis event sources.

**Event source options**

- **Kinesis stream** – The Kinesis stream to read records from.
- **Consumer** (optional) – Use a stream consumer to read from the stream over a dedicated connection.
• **Batch size** – The number of records to send to the function in each batch, up to 10,000. Lambda passes all of the records in the batch to the function in a single call, as long as the total size of the events doesn't exceed the [payload limit](#) for synchronous invocation (6 MB).

• **Batch window** – Specify the maximum amount of time to gather records before invoking the function, in seconds.

• **Starting position** – Process only new records, all existing records, or records created after a certain date.
  - **Latest** – Process new records that are added to the stream.
  - **Trim horizon** – Process all records in the stream.
  - **At timestamp** – Process records starting from a specific time.

After processing any existing records, the function is caught up and continues to process new records.

• **On-failure destination** – A standard SQS queue or standard SNS topic for records that can't be processed. When Lambda discards a batch of records that's too old or has exhausted all retries, Lambda sends details about the batch to the queue or topic.

• **Retry attempts** – The maximum number of times that Lambda retries when the function returns an error. This doesn't apply to service errors or throttles where the batch didn't reach the function.

• **Maximum age of record** – The maximum age of a record that Lambda sends to your function.

• **Split batch on error** – When the function returns an error, split the batch into two before retrying. Your original batch size setting remains unchanged.

• **Concurrent batches per shard** – Concurrently process multiple batches from the same shard.

• **Enabled** – Set to true to enable the event source mapping. Set to false to stop processing records. Lambda keeps track of the last record processed and resumes processing from that point when it's reenabled.

---

**Note**

Kinesis charges for each shard and, for enhanced fan-out, data read from the stream. For pricing details, see [Amazon Kinesis pricing](#).

---

To manage the event source configuration later, choose the trigger in the designer.
Filtering Kinesis events

When you configure Kinesis as an event source for Lambda, you can use event filtering to control which records from your stream Lambda sends to your function for processing. To learn more about using Lambda event filtering with Kinesis, see Filtering with Kinesis.

Event source mapping API

To manage an event source with the AWS Command Line Interface (AWS CLI) or an AWS SDK, you can use the following API operations:

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

To create the event source mapping with the AWS CLI, use the create-event-source-mapping command. The following example uses the AWS CLI to map a function named my-function to a Kinesis data stream. The data stream is specified by an Amazon Resource Name (ARN), with a batch size of 500, starting from the timestamp in Unix time.

```
aws lambda create-event-source-mapping --function-name my-function \
--batch-size 500 --starting-position AT_TIMESTAMP --starting-position-timestamp 1541139109 \
```

You should see the following output:

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "MaximumBatchingWindowInSeconds": 0,
    "ParallelizationFactor": 1,
    "LastModified": 1541139209.351,
    "LastProcessingResult": "No records processed",
}
```
To use a consumer, specify the consumer's ARN instead of the stream's ARN.

Configure additional options to customize how batches are processed and to specify when to discard records that can't be processed. The following example updates an event source mapping to send a failure record to a standard SQS queue after two retry attempts, or if the records are more than an hour old.

```bash
aws lambda update-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b --maximum-retry-attempts 2 --maximum-record-age-in-seconds 3600 --destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-east-2:123456789012:d1q"}}'
```

You should see this output:

```json
{
    "UUID": "f89f8514-cdd9-4602-9e1f-01a5b77d449b",
    "BatchSize": 100,
    "MaximumBatchingWindowInSeconds": 0,
    "ParallelizationFactor": 1,
    "LastModified": 1573243620.0,
    "LastProcessingResult": "PROBLEM: Function call failed",
    "State": "Updating",
    "StateTransitionReason": "User action",
    "DestinationConfig": {},
    "MaximumRecordAgeInSeconds": 604800,
    "BisectBatchOnFunctionError": false,
    "MaximumRetryAttempts": 10000
}
```

Updated settings are applied asynchronously and aren't reflected in the output until the process completes. Use the `get-event-source-mapping` command to view the current status.
aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b

You should see this output:

```
{
    "UUID": "f89f8514-cdd9-4602-9e1f-01a5b77d449b",
    "BatchSize": 100,
    "MaximumBatchingWindowInSeconds": 0,
    "ParallelizationFactor": 1,
    "LastModified": 1573244760.0,
    "LastProcessingResult": "PROBLEM: Function call failed",
    "State": "Enabled",
    "StateTransitionReason": "User action",
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "arn:aws:sqs:us-east-2:123456789012:dlq"
        }
    },
    "MaximumRecordAgeInSeconds": 3600,
    "BisectBatchOnFunctionError": false,
    "MaximumRetryAttempts": 2
}
```

To process multiple batches concurrently, use the `--parallelization-factor` option.

aws lambda update-event-source-mapping --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284 \  
--parallelization-factor 5

**Error handling**

The event source mapping that reads records from your Kinesis stream, invokes your function synchronously, and retries on errors. If Lambda throttles the function or returns an error without invoking the function, Lambda retries until the records expire or exceed the maximum age that you configure on the event source mapping.

If the function receives the records but returns an error, Lambda retries until the records in the batch expire, exceed the maximum age, or reach the configured retry quota. For function errors, you can also configure the event source mapping to split a failed batch into two batches. Retrying
with smaller batches isolates bad records and works around timeout issues. Splitting a batch does not count towards the retry quota.

If the error handling measures fail, Lambda discards the records and continues processing batches from the stream. With the default settings, this means that a bad record can block processing on the affected shard for up to one week. To avoid this, configure your function's event source mapping with a reasonable number of retries and a maximum record age that fits your use case.

To retain a record of discarded batches, configure a failed-event destination. Lambda sends a document to the destination queue or topic with details about the batch.

**To configure a destination for failed-event records**

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Under **Function overview**, choose **Add destination**.
4. For **Source**, choose **Stream invocation**.
5. For **Stream**, choose a stream that is mapped to the function.
6. For **Destination type**, choose the type of resource that receives the invocation record.
7. For **Destination**, choose a resource.
8. Choose **Save**.

The following example shows an invocation record for a Kinesis stream.

**Example invocation record**

```json
{
    "requestContext": {
        "requestId": "c9b8fa9f-5a7f-xmpl-af9c-0c604cde93a5",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    }
}
```
You can use this information to retrieve the affected records from the stream for troubleshooting. The actual records aren't included, so you must process this record and retrieve them from the stream before they expire and are lost.

**Amazon CloudWatch metrics**

Lambda emits the `IteratorAge` metric when your function finishes processing a batch of records. The metric indicates how old the last record in the batch was when processing finished. If your function is processing new events, you can use the iterator age to estimate the latency between when a record is added and when the function processes it.

An increasing trend in iterator age can indicate issues with your function. For more information, see [Working with Lambda function metrics](#).

**Time windows**

Lambda functions can run continuous stream processing applications. A stream represents unbounded data that flows continuously through your application. To analyze information from this continuously updating input, you can bound the included records using a window defined in terms of time.

Tumbling windows are distinct time windows that open and close at regular intervals. By default, Lambda invocations are stateless—you cannot use them for processing data across multiple continuous invocations without an external database. However, with tumbling windows, you can maintain your state across invocations. This state contains the aggregate result of the messages.
previously processed for the current window. Your state can be a maximum of 1 MB per shard. If it exceeds that size, Lambda terminates the window early.

Each record in a stream belongs to a specific window. Lambda will process each record at least once, but doesn't guarantee that each record will be processed only once. In rare cases, such as error handling, some records might be processed more than once. Records are always processed in order the first time. If records are processed more than once, they might be processed out of order.

**Aggregation and processing**

Your user managed function is invoked both for aggregation and for processing the final results of that aggregation. Lambda aggregates all records received in the window. You can receive these records in multiple batches, each as a separate invocation. Each invocation receives a state. Thus, when using tumbling windows, your Lambda function response must contain a `state` property. If the response does not contain a `state` property, Lambda considers this a failed invocation. To satisfy this condition, your function can return a `TimeWindowEventResponse` object, which has the following JSON shape:

**Example TimeWindowEventResponse values**

```json
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

**Note**

For Java functions, we recommend using a `Map<String, String>` to represent the state.

At the end of the window, the flag `isFinalInvokeForWindow` is set to `true` to indicate that this is the final state and that it’s ready for processing. After processing, the window completes and your final invocation completes, and then the state is dropped.

At the end of your window, Lambda uses final processing for actions on the aggregation results. Your final processing is synchronously invoked. After successful invocation, your function
checkpoints the sequence number and stream processing continues. If invocation is unsuccessful, your Lambda function suspends further processing until a successful invocation.

**Example KinesisTimeWindowEvent**

```json
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1607497475.000
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID": "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-kinesis-role",
            "awsRegion": "us-east-1",
            "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream"
        }
    ],
    "window": {
        "start": "2020-12-09T07:04:00Z",
        "end": "2020-12-09T07:06:00Z"
    },
    "state": {
        "1": 282,
        "2": 715
    },
    "shardId": "shardId-000000000006",
    "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```
Configuration

You can configure tumbling windows when you create or update an event source mapping. To configure a tumbling window, specify the window in seconds. The following example AWS Command Line Interface (AWS CLI) command creates a streaming event source mapping that has a tumbling window of 120 seconds. The Lambda function defined for aggregation and processing is named `tumbling-window-example-function`.

```
```

Lambda determines tumbling window boundaries based on the time when records were inserted into the stream. All records have an approximate timestamp available that Lambda uses in boundary determinations.

Tumbling window aggregations do not support resharding. When the shard ends, Lambda considers the window closed, and the child shards start their own window in a fresh state.

Tumbling windows fully support the existing retry policies `maxRetryAttempts` and `maxRecordAge`.

Example Handler.py – Aggregation and processing

The following Python function demonstrates how to aggregate and then process your final state:

```python
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

    # Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

    # Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')
```

Time windows
```python
# Aggregation logic
state = event['state']
for record in event['Records']:
    state[record['kinesis']['partitionKey']] = state.get(record['kinesis']['partitionKey'], 0) + 1

print('Returning state: ', state)
return {'state': state}
```

## Reporting batch item failures

When consuming and processing streaming data from an event source, by default Lambda checkpoints to the highest sequence number of a batch only when the batch is a complete success. Lambda treats all other results as a complete failure and retries processing the batch up to the retry limit. To allow for partial successes while processing batches from a stream, turn on `ReportBatchItemFailures`. Allowing partial successes can help to reduce the number of retries on a record, though it doesn’t entirely prevent the possibility of retries in a successful record.

To turn on `ReportBatchItemFailures`, include the enum value `ReportBatchItemFailures` in the `FunctionResponseTypes` list. This list indicates which response types are enabled for your function. You can configure this list when you create or update an event source mapping.

### Report syntax

When configuring reporting on batch item failures, the `StreamsEventResponse` class is returned with a list of batch item failures. You can use a `StreamsEventResponse` object to return the sequence number of the first failed record in the batch. You can also create your own custom class using the correct response syntax. The following JSON structure shows the required response syntax:

```json
{
    "batchItemFailures": [
        {
            "itemIdentifier": "<id>"
        }
    ]
}
```
## Note

If the `batchItemFailures` array contains multiple items, Lambda uses the record with the lowest sequence number as the checkpoint. Lambda then retries all records starting from that checkpoint.

## Success and failure conditions

Lambda treats a batch as a complete success if you return any of the following:

- An empty `batchItemFailure` list
- A null `batchItemFailure` list
- An empty `EventResponse`
- A null `EventResponse`

Lambda treats a batch as a complete failure if you return any of the following:

- An empty string `itemIdentifier`
- A null `itemIdentifier`
- An `itemIdentifier` with a bad key name

Lambda retries failures based on your retry strategy.

## Bisecting a batch

If your invocation fails and `BisectBatchOnFunctionError` is turned on, the batch is bisected regardless of your `ReportBatchItemFailures` setting.

When a partial batch success response is received and both `BisectBatchOnFunctionError` and `ReportBatchItemFailures` are turned on, the batch is bisected at the returned sequence number and Lambda retries only the remaining records.

Here are some examples of function code that return the list of failed message IDs in the batch:
Reporting Kinesis batch item failures with Lambda using .NET.

```csharp
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.

namespace KinesisIntegration;

class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
```
Logger.LogInformation($"Processed Event with EventId: {record.EventId}");
string data = await GetRecordDataAsync(record.Kinesis, context);
Logger.LogInformation($"Data: {data}");
// TODO: Do interesting work based on the new data
}
catch (Exception ex)
{
    Logger.LogError($"An error occurred {ex.Message}");
    /* Since we are working with streams, we can return the failed item immediately.
    Lambda will immediately begin to retry processing from this failed item onwards. */
    return new StreamsEventResponse
    {
        BatchItemFailures = new
        List<StreamsEventResponse.BatchItemFailure>
        {
            new StreamsEventResponse.BatchItemFailure
            {
                ItemIdentifier = record.Kinesis.SequenceNumber
            }
        }
    }
}
Logger.LogInformation($"Successfully processed {evnt.Records.Count} records.");
return new StreamsEventResponse();

private async Task<string> GetRecordDataAsync(KinesisEvent.Record record, ILambdaContext context)
{
    byte[] bytes = record.Data.ToArray();
    string data = Encoding.UTF8.GetString(bytes);
    await Task.CompletedTask; //Placeholder for actual async work
    return data;
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }

    public class BatchItemFailure
    {
    }
}

Reporting Kinesis batch item failures with Lambda using Java.

```java
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent, StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new ArrayList<>();
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord : input.getRecords()) {
            try {
                //Process your record
```
```javascript
KinesisEvent.Record kinesisRecord =
kinesisEventRecord.getKinesis();
curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

} catch (Exception e) {
    /* Since we are working with streams, we can return the failed
     * item immediately.
     * Lambda will immediately begin to retry processing from this
     * failed item onwards. */
    batchItemFailures.add(new
    StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
    return new StreamsEventResponse(batchItemFailures);
}
return new StreamsEventResponse(batchItemFailures);
```

**JavaScript**

**SDK for JavaScript (v2)**

*Note*  
There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

**Reporting Kinesis batch item failures with Lambda using Javascript.**

```javascript
exports.handler = async (event, context) => {
    for (const record of event.Records) {
        try {
            console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
            const recordData = await getRecordDataAsync(record.kinesis);
            console.log(`Record Data: ${recordData}`);
            // TODO: Do interesting work based on the new data
        } catch (err) {
            console.error(`An error occurred ${err}`);
        }
    }
```
Since we are working with streams, we can return the failed item immediately. Lambda will immediately begin to retry processing from this failed item onwards. */

```javascript
return {
    batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
};
```

```javascript
console.log(`Successfully processed ${event.Records.length} records.`);
return { batchItemFailures: [] };;
```

```javascript
async function getRecordDataAsync(payload) {
    var data = Buffer.from(payload.data, "base64").toString("utf-8");
    await Promise.resolve(1); //Placeholder for actual async work
    return data;
    }
```

Reporting Kinesis batch item failures with Lambda using TypeScript.

```javascript
import {
    KinesisStreamEvent,
    Context,
    KinesisStreamHandler,
    KinesisStreamRecordPayload,
    KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
    logLevel: "INFO",
    serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (event: KinesisStreamEvent,
    context: Context
): Promise<KinesisStreamBatchResponse> => {
    for (const record of event.Records) {
        try {

            Reporting batch item failures
```
logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
const recordData = await getRecordDataAsync(record.kinesis);
logger.info(`Record Data: ${recordData}`);
// TODO: Do interesting work based on the new data
} catch (err) {
    logger.error(`An error occurred ${err}`);
    /* Since we are working with streams, we can return the failed item immediately.
    Lambda will immediately begin to retry processing from this failed item onwards. */
    return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
    };
}
logger.info(`Successfully processed ${event.Records.length} records.`);
return { batchItemFailures: [] };
}
async function getRecordDataAsync(
    payload: KinesisStreamRecordPayload
): Promise<string> {
    var data = Buffer.from(payload.data, "base64").toString("utf-8");
    await Promise.resolve(1); //Placeholder for actual async work
    return data;
}

Python

**SDK for Python (Boto3)**

```

```

### Note

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Python.

```
def handler(event, context):
    records = event.get("Records")
```

```
curRecordSequenceNumber = ""

for record in records:
    try:
        # Process your record
        curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
    except Exception as e:
        # Return failed record's sequence number
        return {"batchItemFailures": [{"itemIdentifier": curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}  

Amazon Kinesis configuration parameters

All Lambda event source types share the same CreateEventSourceMapping and UpdateEventSourceMapping API operations. However, only some of the parameters apply to Kinesis.

Event source parameters that apply to Kinesis

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Default</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BatchSize</td>
<td>N</td>
<td>100</td>
<td>Maximum: 10,000</td>
</tr>
<tr>
<td>BisectBatchOnError</td>
<td>N</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>DestinationConfig</td>
<td>N</td>
<td></td>
<td>standard Amazon SQS queue or standard Amazon SNS topic destination for discarded records</td>
</tr>
<tr>
<td>Enabled</td>
<td>N</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>EventSourceArn</td>
<td>Y</td>
<td></td>
<td>ARN of the data stream or a stream consumer</td>
</tr>
<tr>
<td>Parameter</td>
<td>Required</td>
<td>Default</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>----------</td>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>FunctionName</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaximumBatchingWindowInSeconds</td>
<td>N</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>MaximumRecordAgeInSeconds</td>
<td>N</td>
<td>-1</td>
<td>-1 means infinite: Lambda doesn't discard records</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Minimum: -1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Maximum: 604,800</td>
</tr>
<tr>
<td>MaximumRetryAttempts</td>
<td>N</td>
<td>-1</td>
<td>-1 means infinite: failed records are retried until the record expires</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Minimum: -1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Maximum: 10,000</td>
</tr>
<tr>
<td>ParallelizationFactor</td>
<td>N</td>
<td>1</td>
<td>Maximum: 10</td>
</tr>
<tr>
<td>StartingPosition</td>
<td>Y</td>
<td></td>
<td>AT_TIMESTAMP, TRIM_HORIZON, or LATEST</td>
</tr>
<tr>
<td>StartingPositionTimestamp</td>
<td>N</td>
<td></td>
<td>Only valid if StartingPosition is set to AT_TIMESTAMP. The time from which to start reading, in Unix time seconds</td>
</tr>
</tbody>
</table>
### Tutorial: Using AWS Lambda with Amazon Kinesis

In this tutorial, you create a Lambda function to consume events from a Kinesis stream.

1. Custom app writes records to the stream.
2. AWS Lambda polls the stream and, when it detects new records in the stream, invokes your Lambda function.
3. AWS Lambda runs the Lambda function by assuming the execution role you specified at the time you created the Lambda function.

#### Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console](#) to create your first Lambda function.

To complete the following steps, you need the [AWS Command Line Interface (AWS CLI) version 2](#). Commands and the expected output are listed in separate blocks:

```bash
aws --version
```

You should see the following output:

```bash
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.
Create the execution role

Create the execution role that gives your function permission to access AWS resources.

To create an execution role

1. Open the roles page in the IAM console.
2. Choose Create role.
3. Create a role with the following properties.
   - Trusted entity – AWS Lambda.
   - Permissions – AWSLambdaKinesisExecutionRole.
   - Role name – lambda-kinesis-role.

The AWSLambdaKinesisExecutionRole policy has the permissions that the function needs to read items from Kinesis and write logs to CloudWatch Logs.

Create the function

Create a Lambda function that processes your Kinesis messages. The function code logs the event ID and event data of the Kinesis record to CloudWatch Logs.

This tutorial uses the Node.js 18.x runtime, but we've also provided example code in other runtime languages. You can select the tab in the following box to see code for the runtime you're interested in. The JavaScript code you'll use in this step is in the first example shown in the JavaScript tab.
Consuming a Kinesis event with Lambda using .NET.

```csharp
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
[assembly:
    LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer))]

namespace KinesisIntegrationSampleCode;
public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }
        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation("Processed Event with EventId: {record.EventId}";
```
AWS Lambda

Developer Guide

string data = await GetRecordDataAsync(record.Kinesis, context);
Logger.LogInformation($"Data: {data}");
// TODO: Do interesting work based on the new data
}
catch (Exception ex)
{
Logger.LogError($"An error occurred {ex.Message}");
throw;
}
}
Logger.LogInformation($"Successfully processed {evnt.Records.Count}
records.");
}
private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
ILambdaContext context)
{
byte[] bytes = record.Data.ToArray();
string data = Encoding.UTF8.GetString(bytes);
await Task.CompletedTask; //Placeholder for actual async work
return data;
}
}

Go
SDK for Go V2
Note
There's more on GitHub. Find the complete example and learn how to set up and run
in the Serverless examples repository.

Consuming a Kinesis event with Lambda using Go.
package main
import (
"context"
"log"

Tutorial

1182


func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
    if len(kinesisEvent.Records) == 0 {
        log.Printf("empty Kinesis event received")
        return nil
    }

    for _, record := range kinesisEvent.Records {
        log.Printf("processed Kinesis event with EventId: %v", record.EventID)
        recordDataBytes := record.Kinesis.Data
        recordDataText := string(recordDataBytes)
        log.Printf("record data: %v", recordDataText)
        // TODO: Do interesting work based on the new data
    }
    log.Printf("successfully processed %v records", len(kinesisEvent.Records))
    return nil
}

func main() {
    lambda.Start(handler)
}
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+" records");
        return null;
    }
}

JavaScript

**SDK for JavaScript (v2)**

ℹ️ Note

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using JavaScript.
exports.handler = async (event, context) => {
    for (const record of event.Records) {
        try {
            console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
            const recordData = await getRecordDataAsync(record.kinesis);
            console.log(`Record Data: ${recordData}`);
            // TODO: Do interesting work based on the new data
        } catch (err) {
            console.error(`An error occurred ${err}`);
            throw err;
        }
    }
    console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
    var data = Buffer.from(payload.data, "base64").toString("utf-8");
    await Promise.resolve(1); //Placeholder for actual async work
    return data;
}

Consuming a Kinesis event with Lambda using TypeScript.

import { KinesisStreamEvent, Context, KinesisStreamHandler, KinesisStreamRecordPayload, } from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
    logLevel: "INFO",
    serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (event: KinesisStreamEvent, context: Context): Promise<void> => {
    for (const record of event.Records) {

try {
    logger.info('Processed Kinesis Event - EventID: ${record.eventID}');
    const recordData = await getRecordDataAsync(record.kinesis);
    logger.info('Record Data: ${recordData}');
    // TODO: Do interesting work based on the new data
} catch (err) {
    logger.error(`An error occurred ${err}`);
    throw err;
}
logger.info(`Successfully processed ${event.Records.length} records.`);
}

async function getRecordDataAsync(
    payload: KinesisStreamRecordPayload
): Promise<string> {
    var data = Buffer.from(payload.data, "base64").toString("utf-8");
    await Promise.resolve(1); //Placeholder for actual async work
    return data;
}

Python

**SDK for Python (Boto3)**

![Note]

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

**Consuming a Kinesis event with Lambda using Python.**

```python
import base64
def lambda_handler(event, context):
    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']['data']).decode('utf-8')
```
To create the function

1. Create a directory for the project, and then switch to that directory.
   ```bash
   mkdir kinesis-tutorial
cd kinesis-tutorial
   ```

2. Copy the sample JavaScript code into a new file named `index.js`.

3. Create a deployment package.
   ```bash
   zip function.zip index.js
   ```

4. Create a Lambda function with the `create-function` command.
   ```bash
   ```

Test the Lambda function

Invoke your Lambda function manually using the `invoke` AWS Lambda CLI command and a sample Kinesis event.

To test the Lambda function

1. Copy the following JSON into a file and save it as `input.txt`.
   ```json
   {
     "Records": [
       {
         "kinesis": {
           "kinesisSchemaVersion": "1.0",
           "data": "{record_data}"}
       }
     ]
   }
   ```
2. Use the invoke command to send the event to the function.

    aws lambda invoke --function-name ProcessKinesisRecords \
    --cli-binary-format raw-in-base64-out \
    --payload file://input.txt outputfile.txt

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-commands.html) in the *AWS Command Line Interface User Guide for Version 2*.

The response is saved to `out.txt`.

### Create a Kinesis stream

Use the `create-stream` command to create a stream.

    aws kinesis create-stream --stream-name lambda-stream --shard-count 1

Run the following `describe-stream` command to get the stream ARN.
aws kinesis describe-stream --stream-name lambda-stream

You should see the following output:

```
{
    "StreamDescription": {
        "Shards": [
            {
                "ShardId": "shardId-000000000000",
                "HashKeyRange": {
                    "StartingHashKey": "0",
                    "EndingHashKey": "340282366920746074317682119384634633455"
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber": "4959107394776892513481539594623130411957558361251844610"
                }
            }
        ],
        "StreamName": "lambda-stream",
        "StreamStatus": "ACTIVE",
        "RetentionPeriodHours": 24,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
        "KeyId": null,
        "StreamCreationTimestamp": 1544828156.0
    }
}
```

You use the stream ARN in the next step to associate the stream with your Lambda function.

**Add an event source in AWS Lambda**

Run the following AWS CLI add-event-source command.

```
```
Note the mapping ID for later use. You can get a list of event source mappings by running the `list-event-source-mappings` command.

```
aws lambda list-event-source-mappings --function-name ProcessKinesisRecords \
```

In the response, you can verify the status value is enabled. Event source mappings can be disabled to pause polling temporarily without losing any records.

**Test the setup**

To test the event source mapping, add event records to your Kinesis stream. The `--data` value is a string that the CLI encodes to base64 prior to sending it to Kinesis. You can run the same command more than once to add multiple records to the stream.

```
aws kinesis put-record --stream-name lambda-stream --partition-key 1 \
--data "Hello, this is a test."
```

Lambda uses the execution role to read records from the stream. Then it invokes your Lambda function, passing in batches of records. The function decodes data from each record and logs it, sending the output to CloudWatch Logs. View the logs in the CloudWatch console.

**Clean up your resources**

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

**To delete the execution role**

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose Delete.
4. Enter the name of the role in the text input field and choose Delete.
To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose Actions, Delete.
4. Type delete in the text input field and choose Delete.

To delete the Kinesis stream

1. Sign in to the AWS Management Console and open the Kinesis console at https://console.aws.amazon.com/kinesis.
2. Select the stream you created.
3. Choose Actions, Delete.
4. Enter delete in the text input field.
5. Choose Delete.

Sample function code

To process events from Amazon Kinesis, iterate through the records included in the event object and decode the Base64-encoded data included in each.

Note

The code on this page does not support aggregated records. You can disable aggregation in the Kinesis Producer Library configuration, or use the Kinesis Record Aggregation library to deaggregate records.

Sample code is available for the following languages.

Topics

- Node.js 12.x
- Java 11
- C#
- Python 3
Node.js 12.x

The following example code receives a Kinesis event input and processes the messages that it contains. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Example index.js

```javascript
console.log('Loading function');

exports.handler = function(event, context) {
    //console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        // Kinesis data is base64 encoded so decode here
        var payload = Buffer.from(record.kinesis.data, 'base64').toString('ascii');
        console.log('Decoded payload:', payload);
    });
};
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Node.js Lambda functions with .zip file archives](#).

Java 11

The following is example Java code that receives Kinesis event record data as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

In the code, `recordHandler` is the handler. The handler uses the predefined `KinesisEvent` class that is defined in the `aws-lambda-java-events` library.

Example ProcessKinesisEvents.java

```java
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent.KinesisEventRecord;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent, Void>{
    @Override
```
```java
public void handleRequest(KinesisEvent event, Context context)
{
    for(KinesisEventRecord rec : event.getRecords()) {
        System.out.println(new String(rec.getKinesis().getData().array()));
    }
    return null;
}
```

If the handler returns normally without exceptions, Lambda considers the input batch of records as processed successfully and begins reading new records in the stream. If the handler throws an exception, Lambda considers the input batch of records as not processed and invokes the function with the same batch of records again.

**Dependencies**

- aws-lambda-java-core
- aws-lambda-java-events
- aws-java-sdk

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [Deploy Java Lambda functions with .zip or JAR file archives](#).

**C#**

The following is example C# code that receives Kinesis event record data as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

In the code, `HandleKinesisRecord` is the handler. The handler uses the predefined `KinesisEvent` class that is defined in the `Amazon.Lambda.KinesisEvents` library.

**Example ProcessingKinesisEvents.cs**

```csharp
using System;
using System.IO;
using System.Text;

using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;

namespace KinesisStreams
```
Replace the Program.cs in a .NET Core project with the above sample. For instructions, see Build and deploy C# Lambda functions with .zip file archives.

Python 3

The following is example Python code that receives Kinesis event record data as input and processes it. For illustration, the code writes to some of the incoming event data to CloudWatch Logs.

Example ProcessKinesisRecords.py

```python
from __future__ import print_function
```
```python
# Import json and base64
import json
import base64

def lambda_handler(event, context):
    for record in event['Records']:
        # Kinesis data is base64 encoded so decode here
        payload = base64.b64decode(record['kinesis']['data'])
        print("Decoded payload: ", str(payload))
```
Below is a sample AWS SAM template for the Lambda application from the tutorial. The function and handler in the template are for the Node.js code. If you use a different code sample, update the values accordingly.

**Example template.yaml - Kinesis stream**

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  LambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs18.x
      Timeout: 10
      Tracing: Active
      Events:
        Stream:
          Type: Kinesis
          Properties:
            Stream: !GetAtt stream.Arn
            BatchSize: 100
            StartingPosition: LATEST
  stream:
    Type: AWS::Kinesis::Stream
    Properties:
      ShardCount: 1

Outputs:
  FunctionName:
    Description: "Function name"
    Value: !Ref LambdaFunction
  StreamARN:
    Description: "Stream ARN"
    Value: !GetAtt stream.Arn
```

The template creates a Lambda function, a Kinesis stream, and an event source mapping. The event source mapping reads from the stream and invokes the function.

To use an HTTP/2 stream consumer, create the consumer in the template and configure the event source mapping to read from the consumer instead of from the stream.
Example template.yaml - Kinesis stream consumer

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: A function that processes data from a Kinesis stream.
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs12.x
      Timeout: 10
      Tracing: Active
    Events:
      Stream:
        Type: Kinesis
        Properties:
          Stream: !GetAtt streamConsumer.ConsumerARN
          StartingPosition: LATEST
          BatchSize: 100
  stream:
    Type: "AWS::Kinesis::Stream"
    Properties:
      ShardCount: 1
  streamConsumer:
    Type: "AWS::Kinesis::StreamConsumer"
    Properties:
      StreamARN: !GetAtt stream.Arn
      ConsumerName: "TestConsumer"

Outputs:
  FunctionName:
    Description: "Function name"
    Value: !Ref function
  StreamARN:
    Description: "Stream ARN"
    Value: !GetAtt stream.Arn
  ConsumerARN:
    Description: "Stream consumer ARN"
    Value: !GetAtt streamConsumer.ConsumerARN
```

For information on how to package and deploy your serverless application using the package and deploy commands, see Deploying serverless applications in the AWS Serverless Application Model Developer Guide.
Using Lambda with Kubernetes

You can deploy and manage Lambda functions with the Kubernetes API using AWS Controllers for Kubernetes (ACK) or Crossplane.

AWS Controllers for Kubernetes (ACK)

You can use ACK to deploy and manage AWS resources from the Kubernetes API. Through ACK, AWS provides open-source custom controllers for AWS services such as Lambda, Amazon Elastic Container Registry (Amazon ECR), Amazon Simple Storage Service (Amazon S3), and Amazon SageMaker. Each supported AWS service has its own custom controller. In your Kubernetes cluster, install a controller for each AWS service that you want to use. Then, create a Custom Resource Definition (CRD) to define the AWS resources.

We recommend that you use Helm 3.8 or later to install ACK controllers. Every ACK controller comes with its own Helm chart, which installs the controller, CRDs, and Kubernetes RBAC rules. For more information, see Install an ACK Controller in the ACK documentation.

After you create the ACK custom resource, you can use it like any other built-in Kubernetes object. For example, you can deploy and manage Lambda functions with your preferred Kubernetes toolchains, including kubectl.

Here are some example use cases for provisioning Lambda functions through ACK:

- Your organization uses role-based access control (RBAC) and IAM roles for service accounts to create permissions boundaries. With ACK, you can reuse this security model for Lambda without having to create new users and policies.
- Your organization has a DevOps process to deploy resources into an Amazon Elastic Kubernetes Service (Amazon EKS) cluster using Kubernetes manifests. With ACK, you can use a manifest to provision Lambda functions without creating separate infrastructure as code templates.

For more information about using ACK, see the Lambda tutorial in the ACK documentation.

Crossplane

Crossplane is an open-source Cloud Native Computing Foundation (CNCF) project that uses Kubernetes to manage cloud infrastructure resources. With Crossplane, developers can request infrastructure without needing to understand its complexities. Platform teams retain control over how the infrastructure is provisioned and managed.
Using Crossplane, you can deploy and manage Lambda functions with your preferred Kubernetes toolchains such as `kubectl`, and any CI/CD pipeline that can deploy manifests to Kubernetes. Here are some example use cases for provisioning Lambda functions through Crossplane:

- Your organization wants to enforce compliance by ensuring that Lambda functions have the correct tags. Platform teams can use Crossplane Compositions to define this policy through API abstractions. Developers can then use these abstractions to deploy Lambda functions with tags.

- Your project uses GitOps with Kubernetes. In this model, Kubernetes continuously reconciles the git repository (desired state) with the resources running inside the cluster (current state). If there are differences, the GitOps process automatically makes changes to the cluster. You can use GitOps with Kubernetes for deploying and managing Lambda functions through Crossplane, using familiar Kubernetes tools and concepts such as CRDs and Controllers.

To learn more about using Crossplane with Lambda, see the following:

- [AWS Blueprints for Crossplane](#): This repository includes examples of how to use Crossplane to deploy AWS resources, including Lambda functions.

  [Note] AWS Blueprints for Crossplane are under active development and should not be used in production.

- [Deploying Lambda with Amazon EKS and Crossplane](#): This video demonstrates an advanced example of deploying an AWS serverless architecture with Crossplane, exploring the design from both the developer and platform perspectives.
Using AWS Lambda with Amazon Lex

You can use Amazon Lex to integrate a conversation bot into your application. The Amazon Lex bot provides a conversational interface with your users. Amazon Lex provides prebuilt integration with Lambda, which enables you to use a Lambda function with your Amazon Lex bot.

When you configure an Amazon Lex bot, you can specify a Lambda function to perform validation, fulfillment, or both. For validation, Amazon Lex invokes the Lambda function after each response from the user. The Lambda function can validate the response and provide corrective feedback to the user, if necessary. For fulfillment, Amazon Lex invokes the Lambda function to fulfill the user request after the bot successfully collects all of the required information and receives confirmation from the user.

You can manage the concurrency of your Lambda function to control the maximum number of simultaneous bot conversations that you serve. The Amazon Lex API returns an HTTP 429 status code (Too Many Requests) if the function is at maximum concurrency.

The API returns an HTTP 424 status code (Dependency Failed Exception) if the Lambda function throws an exception.

The Amazon Lex bot invokes your Lambda function synchronously. The event parameter contains information about the bot and the value of each slot in the dialog. For definitions of the event and response fields, see Lambda event and response format in the Amazon Lex Developer Guide. The invocationSource parameter in the Amazon Lex message event indicates whether the Lambda function should validate the inputs (DialogCodeHook) or fulfill the intent (FulfillmentCodeHook).

For an example tutorial that shows how to use Lambda with Amazon Lex, see Exercise 1: Create Amazon Lex bot using a blueprint in the Amazon Lex Developer Guide.

Roles and permissions

You need to configure a service-linked role as your function's execution role. Amazon Lex defines the service-linked role with predefined permissions. When you create an Amazon Lex bot using the console, the service-linked role is created automatically. To create a service-linked role with the AWS CLI, use the create-service-linked-role command.

```
aws iam create-service-linked-role --aws-service-name lex.amazonaws.com
```

This command creates the following role.
If your Lambda function uses other AWS services, you need to add the corresponding permissions to the service-linked role.

You use a resource-based permissions policy to allow the Amazon Lex bot to invoke your Lambda function. If you use the Amazon Lex console, the permissions policy is created automatically. From the AWS CLI, use the Lambda `add-permission` command to set the permission.

For Amazon Lex V2, run the following command. In the source ARN, replace `us-east-1` with the AWS Region your Amazon Lex bot is in, and use your own AWS account number and bot alias.

```bash
aws lambda add-permission \
  --function-name LexCodeHook \
  --statement-id LexInvoke-MyBot \
  --action lambda:InvokeFunction \
  --principal lex.amazonaws.com \
  --source-arn "arn:aws:lex:us-east-1:123456789012:bot-alias/MYBOT/MYBOTALIAS"
```
You can also use Amazon Lex V1 to invoke a Lambda function. For Amazon Lex V1, run the following command. In the source ARN, replace `us-east-1` with the AWS Region your Amazon Lex intent is in and use your own AWS account number and intent name.

```bash
aws lambda add-permission \
  --function-name LexCodeHook \
  --statement-id LexInvoke-MyIntent \
  --action lambda:InvokeFunction \
  --principal lex.amazonaws.com \
  --source-arn "arn:aws:lex:us-east-1:123456789012 ID:intent:MYINTENT:*"
```

Note that Amazon Lex V1 is no longer maintained. We recommend that you use Amazon Lex V2.
Using Lambda with Amazon MQ

Note

If you want to send data to a target other than a Lambda function or enrich the data before sending it, see Amazon EventBridge Pipes.

Amazon MQ is a managed message broker service for Apache ActiveMQ and RabbitMQ. A message broker enables software applications and components to communicate using various programming languages, operating systems, and formal messaging protocols through either topic or queue event destinations.

Amazon MQ can also manage Amazon Elastic Compute Cloud (Amazon EC2) instances on your behalf by installing ActiveMQ or RabbitMQ brokers and by providing different network topologies and other infrastructure needs.

You can use a Lambda function to process records from your Amazon MQ message broker. Lambda invokes your function through an event source mapping, a Lambda resource that reads messages from your broker and invokes the function synchronously.

The Amazon MQ event source mapping has the following configuration restrictions:

- Concurrency – Lambda functions that use an Amazon MQ event source mapping have a default maximum concurrency setting. For ActiveMQ, the Lambda service limits the number of concurrent execution environments to five. For RabbitMQ, the number of concurrent execution environments is limited to 1. Even if you change your function's reserved or provisioned concurrency settings, the Lambda service won't make more execution environments available. To request an increase in the default maximum concurrency, contact AWS Support.
- Cross account – Lambda does not support cross-account processing. You cannot use Lambda to process records from an Amazon MQ message broker that is in a different AWS account.
- Authentication – For ActiveMQ, only the ActiveMQ SimpleAuthenticationPlugin is supported. For RabbitMQ, only the PLAIN authentication mechanism is supported. Users must use AWS Secrets Manager to manage their credentials. For more information about ActiveMQ authentication, see Integrating ActiveMQ brokers with LDAP in the Amazon MQ Developer Guide.
- Connection quota – Brokers have a maximum number of allowed connections per wire-level protocol. This quota is based on the broker instance type. For more information, see the Brokers section of Quotas in Amazon MQ in the Amazon MQ Developer Guide.
• Connectivity – You can create brokers in a public or private virtual private cloud (VPC). For private VPCs, your Lambda function needs access to the VPC to receive messages. For more information, see the section called “Network configuration” later in this topic.

• Event destinations – Only queue destinations are supported. However, you can use a virtual topic, which behaves as a topic internally while interacting with Lambda as a queue. For more information, see Virtual Destinations on the Apache ActiveMQ website, and Virtual Hosts on the RabbitMQ website.

• Network topology – For ActiveMQ, only one single-instance or standby broker is supported per event source mapping. For RabbitMQ, only one single-instance broker or cluster deployment is supported per event source mapping. Single-instance brokers require a failover endpoint. For more information about these broker deployment modes, see Active MQ Broker Architecture and Rabbit MQ Broker Architecture in the Amazon MQ Developer Guide.

• Protocols – Supported protocols depend on the type of Amazon MQ integration.
  • For ActiveMQ integrations, Lambda consumes messages using the OpenWire/Java Message Service (JMS) protocol. No other protocols are supported for consuming messages. Within the JMS protocol, only TextMessage and BytesMessage are supported. Lambda also supports JMS custom properties. For more information about the OpenWire protocol, see OpenWire on the Apache ActiveMQ website.
  • For RabbitMQ integrations, Lambda consumes messages using the AMQP 0-9-1 protocol. No other protocols are supported for consuming messages. For more information about RabbitMQ’s implementation of the AMQP 0-9-1 protocol, see AMQP 0-9-1 Complete Reference Guide on the RabbitMQ website.

Lambda automatically supports the latest versions of ActiveMQ and RabbitMQ that Amazon MQ supports. For the latest supported versions, see Amazon MQ release notes in the Amazon MQ Developer Guide.

⚠️ Note

By default, Amazon MQ has a weekly maintenance window for brokers. During that window of time, brokers are unavailable. For brokers without standby, Lambda cannot process any messages during that window.

Sections

• Lambda consumer group
Lambda consumer group

To interact with Amazon MQ, Lambda creates a consumer group which can read from your Amazon MQ brokers. The consumer group is created with the same ID as the event source mapping UUID.

For Amazon MQ event sources, Lambda batches records together and sends them to your function in a single payload. To control behavior, you can configure the batching window and batch size. Lambda pulls messages until it processes the payload size maximum of 6 MB, the batching window expires, or the number of records reaches the full batch size. For more information, see Batching behavior.

The consumer group retrieves the messages as a BLOB of bytes, base64-encodes them into a single JSON payload, and then invokes your function. If your function returns an error for any of the messages in a batch, Lambda retries the whole batch of messages until processing succeeds or the messages expire.

Note

While Lambda functions typically have a maximum timeout limit of 15 minutes, event source mappings for Amazon MSK, self-managed Apache Kafka, Amazon DocumentDB, and Amazon MQ for ActiveMQ and RabbitMQ only support functions with maximum timeout limits of 14 minutes. This constraint ensures that the event source mapping can properly handle function errors and retries.

You can monitor a given function's concurrency usage using the ConcurrentExecutions metric in Amazon CloudWatch. For more information about concurrency, see the section called “Configuring reserved concurrency”.

Lambda consumer group
Example Amazon MQ record events

ActiveMQ

```json
{
    "eventSource": "aws:mq",
    "messages": [
        {
            "messageType": "jms/text-message",
            "deliveryMode": 1,
            "replyTo": null,
            "type": null,
            "expiration": "60000",
            "priority": 1,
            "correlationId": "myJMSCoID",
            "redelivered": false,
            "destination": {
                "physicalName": "testQueue"
            },
            "data": "QUJDOkFBQUE=",
            "timestamp": 1598827811958,
            "brokerInTime": 1598827811958,
            "brokerOutTime": 1598827811959,
            "properties": {
                "index": "1",
                "doAlarm": "false",
                "myCustomProperty": "value"
            }
        },
        {
            "messageType": "jms/bytes-message",
            "deliveryMode": 1,
            "replyTo": null,
            "type": null,
            "expiration": "60000",
            "priority": 2,
            "correlationId": "myJMSCoID1",
            "redelivered": false,
```
"destination": {  
  "physicalName": "testQueue"  
},  
"data": "LQaGQ82S48k=",  
"timestamp": 1598827811958,  
"brokerInTime": 1598827811958,  
"brokerOutTime": 1598827811959,  
"properties": {  
  "index": "1",  
  "doAlarm": "false",  
  "myCustomProperty": "value"  
}  
]  
}  
]  
}  
}  
}  
}  
}
In the RabbitMQ example, pizzaQueue is the name of the RabbitMQ queue, and / is the name of the virtual host. When receiving messages, the event source lists messages under pizzaQueue::/.

**Execution role permissions**

To read records from an Amazon MQ broker, your Lambda function needs the following permissions added to its execution role:
Network configuration

By default, when you create an Amazon MQ broker, the PubliclyAccessible flag is set to false. For your broker to receive a public IP address, you must set the PubliclyAccessible flag to true.

To give Lambda full access to your broker through your event source mapping, either your broker must use a public endpoint (public IP address), or you must provide access to the Amazon VPC you created the broker in. Best practice for using Amazon MQ with Lambda is to use private endpoints and to give your Lambda function access to your broker’s VPC. See Prefer brokers without public accessibility in the Amazon MQ Developer Guide for more information.

Note that when you add Amazon MQ as a trigger, Lambda assumes the VPC settings of the Amazon MQ broker, not the Lambda function’s VPC settings. For your Lambda function to access the VPC with your Amazon MQ broker, do one of the following:

- mq:DescribeBroker
- secretsmanager:GetSecretValue
- ec2:CreateNetworkInterface
- ec2:DeleteNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeSecurityGroups
- ec2:DescribeSubnets
- ec2:DescribeVpcs
- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

Note

When using an encrypted customer managed key, add the kms:Decrypt permission as well.
• Configure a NAT gateway on each public subnet in the VPC containing your Amazon MQ broker. For more information, see the section called “Internet and service access for VPC-connected functions”.

• Create a connection between the Amazon VPC containing your Amazon MQ broker and a Lambda VPC endpoint. Your Amazon VPC must also connect to AWS Security Token Service (AWS STS) and Secrets Manager endpoints. For more information, see the section called “Inbound networking”.

You must also configure the following Amazon VPC security group rules on your VPC:

• Inbound rules – Allow all traffic on the broker port for the security group specified for your event source from within its own security group. ActiveMQ uses port 61617 by default and RabbitMQ uses port 5671 by default.

• Outbound rules – Allow all traffic on port 443 for all destinations. Allow all traffic on the broker port for within its own security group. ActiveMQ uses port 61617 by default and RabbitMQ uses port 5671 by default.

If you use VPC endpoints instead of a NAT gateway, the security groups associated with the VPC endpoints must allow all inbound traffic on port 443 from the event source's security groups.

**Configuring a broker as an event source**

Create an [event source mapping](#) to tell Lambda to send records from an Amazon MQ broker to a Lambda function. You can create multiple event source mappings to process the same data with multiple functions, or to process items from multiple sources with a single function.

To configure your function to read from Amazon MQ, create an [MQ trigger](#) in the Lambda console.

**To create a trigger**

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of a function.
4. Choose a trigger type.
5. Configure the required options, and then choose Add.
Lambda supports the following options for Amazon MQ event sources:

- **MQ broker** – Select an Amazon MQ broker.
- **Batch size** – Set the maximum number of messages to retrieve in a single batch.
- **Queue name** – Enter the Amazon MQ queue to consume.
- **Source access configuration** – Enter virtual host information and the Secrets Manager secret that stores your broker credentials.
- **Enable trigger** – Disable the trigger to stop processing records.

To enable or disable the trigger (or delete it), choose the MQ trigger in the designer. To reconfigure the trigger, use the event source mapping API operations.

**Event source mapping API**

To manage an event source with the [AWS Command Line Interface (AWS CLI)](https://aws.amazon.com/cli/) or an [AWS SDK](https://aws.amazon.com/sdk/), you can use the following API operations:

- `CreateEventSourceMapping`
- `ListEventSourceMappings`
- `GetEventSourceMapping`
- `UpdateEventSourceMapping`
- `DeleteEventSourceMapping`

**Note**

When you update, disable, or delete an event source mapping for Amazon MQ, it can take up to 15 minutes for your changes to take effect. Before this period has elapsed, your event source mapping may continue to process events and invoke your function using your previous settings. This is true even when the status of the event source mapping displayed in the console indicates that your changes have been applied.

To create the event source mapping with the AWS Command Line Interface (AWS CLI), use the `create-event-source-mapping` command.
The following example AWS CLI command creates an event source which maps a Lambda function named `MQ-Example-Function` to an Amazon MQ RabbitMQ-based broker named `ExampleMQBroker`. The command also provides the virtual host name and a Secrets Manager secret ARN that stores the broker credentials.

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:mq:us-east-1:123456789012:broker:ExampleMQBroker:b-24cabbb4-b295-49b7-8543-7ce7ce9dfb98 \ 
--function-name arn:aws:lambda:us-east-1:123456789012:function:MQ-Example-Function \ 
--queues ExampleQueue \ 
--source-access-configuration Type=VIRTUAL_HOST,URI="/" \ Type=BASIC_AUTH,URI=arn:aws:secretsmanager:us-east-1:123456789012:secret:ExampleMQBrokerUserPassword-xPBMTt
```

You should see the following output:

```
{
  "UUID": "91eaeb7e-c976-1234-9451-8709db01f137",
  "BatchSize": 100,
  "EventSourceArn": "arn:aws:mq:us-east-1:123456789012:broker:ExampleMQBroker:b-b4d492ef-bdc3-45e3-a781-cd1a3102ecca",
  "LastModified": 1601927898.741,
  "LastProcessingResult": "No records processed",
  "State": "Creating",
  "StateTransitionReason": "USER_INITIATED",
  "Queues": ["ExampleQueue"],
  "SourceAccessConfigurations": [
    {"Type": "BASIC_AUTH",
     "URI": "arn:aws:secretsmanager:us-east-1:123456789012:secret:ExampleMQBrokerUserPassword-xPBMTt"
    }
  ]
}
```

Using the `update-event-source-mapping` command, you can configure additional options such as how Lambda processes batches and to specify when to discard records that cannot be processed. The following example command updates an event source mapping to have a batch size of 2.
aws lambda update-event-source-mapping
--uuid 91eaeb7e-c976-1234-9451-8709db01f137
--batch-size 2

You should see the following output:

```
{
   "UUID": "91eaeb7e-c976-1234-9451-8709db01f137",
   "BatchSize": 2,
   "EventSourceArn": "arn:aws:mq:us-east-1:123456789012:broker:ExampleMQBroker:b-b4d492ef-bdc3-45e3-a781-cd1a3102ecca",
   "LastModified": 1601928393.531,
   "LastProcessingResult": "No records processed",
   "State": "Updating",
   "StateTransitionReason": "USER_INITIATED"
}
```

Lambda updates these settings asynchronously. The output will not reflect changes until this process completes. To view the current status of your resource, use the `get-event-source-mapping` command.

aws lambda get-event-source-mapping
--uuid 91eaeb7e-c976-4939-9451-8709db01f137

You should see the following output:

```
{
   "UUID": "91eaeb7e-c976-4939-9451-8709db01f137",
   "BatchSize": 2,
   "EventSourceArn": "arn:aws:mq:us-east-1:123456789012:broker:ExampleMQBroker:b-b4d492ef-bdc3-45e3-a781-cd1a3102ecca",
   "LastModified": 1601928393.531,
   "LastProcessingResult": "No records processed",
   "State": "Enabled",
   "StateTransitionReason": "USER_INITIATED"
}
```
Event source mapping errors

When a Lambda function encounters an unrecoverable error, your Amazon MQ consumer stops processing records. Any other consumers can continue processing, provided that they do not encounter the same error. To determine the potential cause of a stopped consumer, check the StateTransitionReason field in the return details of your EventSourceMapping for one of the following codes:

**ESM_CONFIG_NOT_VALID**

The event source mapping configuration is not valid.

**EVENT_SOURCE_AUTHN_ERROR**

Lambda failed to authenticate the event source.

**EVENT_SOURCE_AUTHZ_ERROR**

Lambda does not have the required permissions to access the event source.

**FUNCTION_CONFIG_NOT_VALID**

The function's configuration is not valid.

Records also go unprocessed if Lambda drops them due to their size. The size limit for Lambda records is 6 MB. To redeliver messages upon function error, you can use a dead-letter queue (DLQ). For more information, see [Message Redelivery and DLQ Handling](https://activemq.apache.org/redelivery.html) on the Apache ActiveMQ website and [Reliability Guide](https://www.rabbitmq.com/guides/reliability.html) on the RabbitMQ website.

**Note**

Lambda does not support custom redelivery policies. Instead, Lambda uses a policy with the default values from the [Redelivery Policy](https://activemq.apache.org/redelivery.html) page on the Apache ActiveMQ website, with maximumRedeliveries set to 5.

Amazon MQ and RabbitMQ configuration parameters

All Lambda event source types share the same [CreateEventSourceMapping](https://docs.aws.amazon.com/lambda/latest/dg/lambda-event-source-mappings-create.html) and [UpdateEventSourceMapping](https://docs.aws.amazon.com/lambda/latest/dg/lambda-event-source-mappings-update.html) API operations. However, only some of the parameters apply to Amazon MQ and RabbitMQ.
### Event source parameters that apply to Amazon MQ and RabbitMQ

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Default</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BatchSize</td>
<td>N</td>
<td>100</td>
<td>Maximum: 10,000</td>
</tr>
<tr>
<td>Enabled</td>
<td>N</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>FunctionName</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FilterCriteria</td>
<td>N</td>
<td></td>
<td><a href="#">Lambda event filtering</a></td>
</tr>
<tr>
<td>MaximumBatchingWindowInSeconds</td>
<td>N</td>
<td>500 ms</td>
<td><a href="#">Batching behavior</a></td>
</tr>
<tr>
<td>Queues</td>
<td>N</td>
<td></td>
<td>The name of the Amazon MQ broker destination queue to consume.</td>
</tr>
<tr>
<td>SourceAccessConfigurations</td>
<td>N</td>
<td></td>
<td>For ActiveMQ, BASIC_AUTH credentials. For RabbitMQ, can contain both BASIC_AUTH credentials and VIRTUAL_HOST information.</td>
</tr>
</tbody>
</table>

---

[Batching behavior](#): Indicates the way messages are batched and processed by the Lambda function.

[Lambda event filtering](#): Refers to the ability to filter Lambda events based on specific criteria.
Using Lambda with Amazon MSK

Note
If you want to send data to a target other than a Lambda function or enrich the data before sending it, see Amazon EventBridge Pipes.

Amazon Managed Streaming for Apache Kafka (Amazon MSK) is a fully managed service that you can use to build and run applications that use Apache Kafka to process streaming data. Amazon MSK simplifies the setup, scaling, and management of clusters running Kafka. Amazon MSK also makes it easier to configure your application for multiple Availability Zones and for security with AWS Identity and Access Management (IAM). Amazon MSK supports multiple open-source versions of Kafka.

Amazon MSK as an event source operates similarly to using Amazon Simple Queue Service (Amazon SQS) or Amazon Kinesis. Lambda internally polls for new messages from the event source and then synchronously invokes the target Lambda function. Lambda reads the messages in batches and provides these to your function as an event payload. The maximum batch size is configurable (the default is 100 messages). For more information, see Batching behavior.

Lambda reads the messages sequentially for each partition. A single Lambda payload can contain messages from multiple partitions. After Lambda processes each batch, it commits the offsets of the messages in that batch. If your function returns an error for any of the messages in a batch, Lambda retries the whole batch of messages until processing succeeds or the messages expire.

Note
While Lambda functions typically have a maximum timeout limit of 15 minutes, event source mappings for Amazon MSK, self-managed Apache Kafka, Amazon DocumentDB, and Amazon MQ for ActiveMQ and RabbitMQ only support functions with maximum timeout limits of 14 minutes. This constraint ensures that the event source mapping can properly handle function errors and retries.

For an example of how to configure Amazon MSK as an event source, see Using Amazon MSK as an event source for AWS Lambda on the AWS Compute Blog. For a complete tutorial, see Amazon MSK Lambda Integration in the Amazon MSK Labs.
Example event

Lambda sends the batch of messages in the event parameter when it invokes your function. The event payload contains an array of messages. Each array item contains details of the Amazon MSK topic and partition identifier, together with a timestamp and a base64-encoded message.

```json
{
    "eventSource":"aws:kafka",
    "eventSourceArn":"arn:aws:kafka:sa-east-1:123456789012:cluster/vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
    "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-east-1.amazonaws.com:9092",
    "records":{
        "mytopic-0":[
            {
                "topic":"mytopic",
                "partition":0,
                "offset":15,
                "timestamp":1545084650987,
                "timestampType":"CREATE_TIME",
                "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
                "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "headers":[]
            }
        ]
    }
}
```
MSK cluster authentication

Lambda needs permission to access the Amazon MSK cluster, retrieve records, and perform other tasks. Amazon MSK supports several options for controlling client access to the MSK cluster.

Cluster access options

- Unauthenticated access
- SASL/SCRAM authentication
- IAM role-based authentication
- Mutual TLS authentication
- Configuring the mTLS secret
- How Lambda chooses a bootstrap broker

Unauthenticated access

If no clients access the cluster over the internet, you can use unauthenticated access.
SASL/SCRAM authentication

Amazon MSK supports Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) authentication with Transport Layer Security (TLS) encryption. For Lambda to connect to the cluster, you store the authentication credentials (user name and password) in an AWS Secrets Manager secret.

For more information about using Secrets Manager, see User name and password authentication with AWS Secrets Manager in the Amazon Managed Streaming for Apache Kafka Developer Guide.

Amazon MSK doesn’t support SASL/PLAIN authentication.

IAM role-based authentication

You can use IAM to authenticate the identity of clients that connect to the MSK cluster. If IAM auth is active on your MSK cluster, and you don’t provide a secret for auth, Lambda automatically defaults to using IAM auth. To create and deploy user or role-based policies, use the IAM console or API. For more information, see IAM access control in the Amazon Managed Streaming for Apache Kafka Developer Guide.

To allow Lambda to connect to the MSK cluster, read records, and perform other required actions, add the following permissions to your function’s execution role.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeGroup",
                "kafka-cluster:AlterGroup",
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:ReadData",
                "kafka-cluster:DescribeClusterDynamicConfiguration"
            ],
            "Resource": [
                "arn:aws:kafka:region:account-id:group/cluster-name/cluster-uuid/consumer-group-id"
            ]
        }
    ]
}
```
You can scope these permissions to a specific cluster, topic, and group. For more information, see the Amazon MSK Kafka actions in the Amazon Managed Streaming for Apache Kafka Developer Guide.

**Mutual TLS authentication**

Mutual TLS (mTLS) provides two-way authentication between the client and server. The client sends a certificate to the server for the server to verify the client, and the server sends a certificate to the client for the client to verify the server.

For Amazon MSK, Lambda acts as the client. You configure a client certificate (as a secret in Secrets Manager) to authenticate Lambda with the brokers in your MSK cluster. The client certificate must be signed by a CA in the server's trust store. The MSK cluster sends a server certificate to Lambda to authenticate the brokers with Lambda. The server certificate must be signed by a certificate authority (CA) that's in the AWS trust store.

For instructions on how to generate a client certificate, see Introducing mutual TLS authentication for Amazon MSK as an event source.

Amazon MSK doesn't support self-signed server certificates, because all brokers in Amazon MSK use public certificates signed by Amazon Trust Services CAs, which Lambda trusts by default.

For more information about mTLS for Amazon MSK, see Mutual TLS Authentication in the Amazon Managed Streaming for Apache Kafka Developer Guide.

**Configuring the mTLS secret**

The CLIENT_CERTIFICATE_TLS_AUTH secret requires a certificate field and a private key field. For an encrypted private key, the secret requires a private key password. Both the certificate and private key must be in PEM format.

**Note**

Lambda supports the PBES1 (but not PBES2) private key encryption algorithms.

MSK cluster authentication
The certificate field must contain a list of certificates, beginning with the client certificate, followed by any intermediate certificates, and ending with the root certificate. Each certificate must start on a new line with the following structure:

-----BEGIN CERTIFICATE-----
<certificate contents>
-----END CERTIFICATE-----

Secrets Manager supports secrets up to 65,536 bytes, which is enough space for long certificate chains.

The private key must be in **PKCS #8** format, with the following structure:

-----BEGIN PRIVATE KEY-----
<private key contents>
-----END PRIVATE KEY-----

For an encrypted private key, use the following structure:

-----BEGIN ENCRYPTED PRIVATE KEY-----
<private key contents>
-----END ENCRYPTED PRIVATE KEY-----

The following example shows the contents of a secret for mTLS authentication using an encrypted private key. For an encrypted private key, you include the private key password in the secret.

```json
{
  "privateKeyPassword": "testpassword",
  "certificate": "-----BEGIN CERTIFICATE-----
  MIIE5DCCAsygAwIBAgIRAPJdwaFaNRzytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
  ...j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCROpimO40s1cRqtFHXoa10QQbIlxk
  cmUuiAi9R0=
  -----END CERTIFICATE-----
  -----BEGIN CERTIFICATE-----
  MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hfNC5RdfmHzANBgkqhkiG9w0BAQsFADBb
  ...
jQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS9FkXm0vqVD/YpXkWA/no
  c8PH3PSoAaRwMMgOSA2ALjvRz8mpg==
  -----END CERTIFICATE-----",
  "privateKey": "-----BEGIN ENCRYPTED PRIVATE KEY-----
  MIIFKzBVBghkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGQUiAFcK5hT/X7Kjmgp
```
How Lambda chooses a bootstrap broker

Lambda chooses a bootstrap broker based on the authentication methods available on your cluster, and whether you provide a secret for authentication. If you provide a secret for mTLS or SASL/SCRAM, Lambda automatically chooses that auth method. If you don't provide a secret, Lambda selects the strongest auth method that's active on your cluster. The following is the order of priority in which Lambda selects a broker, from strongest to weakest auth:

- mTLS (secret provided for mTLS)
- SASL/SCRAM (secret provided for SASL/SCRAM)
- SASL IAM (no secret provided, and IAM auth active)
- Unauthenticated TLS (no secret provided, and IAM auth not active)
- Plaintext (no secret provided, and both IAM auth and unauthenticated TLS are not active)

**Note**

If Lambda can't connect to the most secure broker type, Lambda doesn't attempt to connect to a different (weaker) broker type. If you want Lambda to choose a weaker broker type, deactivate all stronger auth methods on your cluster.

Managing API access and permissions

In addition to accessing the Amazon MSK cluster, your function needs permissions to perform various Amazon MSK API actions. You add these permissions to the function's execution role. If your users need access to any of the Amazon MSK API actions, add the required permissions to the identity policy for the user or role.

You can add each of the following permissions to your execution role manually. Alternatively, you can attach the AWS managed policy AWSLambdaMSKEExecutionRole to your execution role. The AWSLambdaMSKEExecutionRole policy contains all required API actions and VPC permissions listed below.
Required Lambda function execution role permissions

To create and store logs in a log group in Amazon CloudWatch Logs, your Lambda function must have the following permissions in its execution role:

- `logs:CreateLogGroup`
- `logs:CreateLogStream`
- `logs:PutLogEvents`

For Lambda to access your Amazon MSK cluster on your behalf, your Lambda function must have the following permissions in its execution role:

- `kafka:DescribeCluster`
- `kafka:DescribeClusterV2`
- `kafka:GetBootstrapBrokers`
- `kafka:DescribeVpcConnection`: Only required for cross-account event source mappings.
- `kafka:ListVpcConnections`: Not required in execution role, but required for an IAM principal that is creating a cross-account event source mapping.

You only need to add one of either `kafka:DescribeCluster` or `kafka:DescribeClusterV2`. For provisioned MSK clusters, either permission works. For serverless MSK clusters, you must use `kafka:DescribeClusterV2`.

Note

Lambda eventually plans to remove the `kafka:DescribeCluster` permission from the associated AWSLambdaMSKExecutionRole managed policy. If you use this policy, you should migrate any applications using `kafka:DescribeCluster` to use `kafka:DescribeClusterV2` instead.

VPC permissions

If only users within a VPC can access your Amazon MSK cluster, your Lambda function must have permission to access your Amazon VPC resources. These resources include your VPC, subnets,
security groups, and network interfaces. To access these resources, your function's execution role must have the following permissions:

- `ec2:CreateNetworkInterface`
- `ec2:DescribeNetworkInterfaces`
- `ec2:DescribeVpcs`
- `ec2:DeleteNetworkInterface`
- `ec2:DescribeSubnets`
- `ec2:DescribeSecurityGroups`

**Optional Lambda function permissions**

Your Lambda function might also need permissions to:

- Access your SCRAM secret, if using SASL/SCRAM authentication.
- Describe your Secrets Manager secret.
- Access your AWS Key Management Service (AWS KMS) customer managed key.
- Send records of failed invocations to a destination.

**Secrets Manager and AWS KMS permissions**

Depending on the type of access control that you're configuring for your Amazon MSK brokers, your Lambda function might need permission to access your SCRAM secret (if using SASL/SCRAM authentication), or Secrets Manager secret to decrypt your AWS KMS customer managed key. To access these resources, your function's execution role must have the following permissions:

- `kafka:ListScramSecrets`
- `secretsmanager:GetSecretValue`
- `kms:Decrypt`

**Sending records to a destination**

If you want to send records of failed invocations to an on-failure destination, your Lambda function must have permission to send these records. For Kafka event source mappings, you can choose between an Amazon SNS topic, Amazon SQS queue, or Amazon S3 bucket as the destination. To send records to an SNS topic, your function's execution role must have the following permission:
To send records to an SQS queue, your function's execution role must have the following permission:

- **sns:Publish**

To send records to an S3 bucket, your function's execution role must have the following permissions:

- **s3:PutObject**
- **s3:ListBuckets**

Additionally, if you configured a KMS key on your destination, Lambda needs the following permissions depending on the destination type:

- If you've enabled encryption with your own KMS key for an S3 destination, **kms:GenerateDataKey** is required. If the KMS key and S3 bucket destination are in a different account from your Lambda function and execution role, configure the KMS key to trust the execution role to allow kms:GenerateDataKey.

- If you've enabled encryption with your own KMS key for SQS destination, **kms:Decrypt** and **kms:GenerateDataKey** are required. If the KMS key and SQS queue destination are in a different account from your Lambda function and execution role, configure the KMS key to trust the execution role to allow kms:Decrypt, kms:GenerateDataKey, **kms:DescribeKey**, and **kms:ReEncrypt**.

- If you've enabled encryption with your own KMS key for SNS destination, **kms:Decrypt** and **kms:GenerateDataKey** are required. If the KMS key and SNS topic destination are in a different account from your Lambda function and execution role, configure the KMS key to trust the execution role to allow kms:Decrypt, kms:GenerateDataKey, **kms:DescribeKey**, and **kms:ReEncrypt**.

### Adding permissions to your execution role

Follow these steps to add the AWS managed policy AWSLambdaMSKEExecutionRole to your execution role using the IAM console.
To add an AWS managed policy

1. Open the Policies page of the IAM console.
2. In the search box, enter the policy name (AWSLambdaMSKExecutionRole).
3. Select the policy from the list, and then choose Policy actions, Attach.
4. On the Attach policy page, select your execution role from the list, and then choose Attach policy.

Granting users access with an IAM policy

By default, users and roles don't have permission to perform Amazon MSK API operations. To grant access to users in your organization or account, you can add or update an identity-based policy. For more information, see Amazon MSK Identity-Based Policy Examples in the Amazon Managed Streaming for Apache Kafka Developer Guide.

Authentication and authorization errors

If any of the permissions required to consume data from the Amazon MSK cluster are missing, Lambda displays one of the following error messages in the event source mapping under LastProcessingResult.

Error messages

- Cluster failed to authorize Lambda
- SASL authentication failed
- Server failed to authenticate Lambda
- Provided certificate or private key is invalid

Cluster failed to authorize Lambda

For SASL/SCRAM or mTLS, this error indicates that the provided user doesn't have all of the following required Kafka access control list (ACL) permissions:

- DescribeConfigs Cluster
- Describe Group
- Read Group
• Describe Topic
• Read Topic

For IAM access control, your function's execution role is missing one or more of the permissions required to access the group or topic. Review the list of required permissions in the section called “IAM role-based authentication”.

When you create either Kafka ACLs or an IAM policy with the required Kafka cluster permissions, specify the topic and group as resources. The topic name must match the topic in the event source mapping. The group name must match the event source mapping's UUID.

After you add the required permissions to the execution role, it might take several minutes for the changes to take effect.

**SASL authentication failed**

For SASL/SCRAM, this error indicates that the provided user name and password aren't valid.

For IAM access control, the execution role is missing the kafka-cluster:Connect permission for the MSK cluster. Add this permission to the role and specify the cluster's Amazon Resource Name (ARN) as a resource.

You might see this error occurring intermittently. The cluster rejects connections after the number of TCP connections exceeds the Amazon MSK service quota. Lambda backs off and retries until a connection is successful. After Lambda connects to the cluster and polls for records, the last processing result changes to OK.

**Server failed to authenticate Lambda**

This error indicates that the Amazon MSK Kafka brokers failed to authenticate with Lambda. This can occur for any of the following reasons:

• You didn't provide a client certificate for mTLS authentication.
• You provided a client certificate, but the brokers aren't configured to use mTLS.
• A client certificate isn't trusted by the brokers.
Provided certificate or private key is invalid

This error indicates that the Amazon MSK consumer couldn't use the provided certificate or private key. Make sure that the certificate and key use PEM format, and that the private key encryption uses a PBES1 algorithm.

Network configuration

If you configure Amazon VPC access to your Kafka brokers, Lambda must have access to the Amazon VPC resources your Kafka cluster resides in. We recommend that you deploy AWS PrivateLink VPC endpoints for Lambda and AWS Security Token Service (AWS STS). If the broker uses authentication, also deploy a VPC endpoint for Secrets Manager. If you configured an on-failure destination, also deploy a VPC endpoint for the destination service.

Alternatively, ensure that the VPC associated with your Kafka cluster includes one NAT gateway per public subnet. For more information, see Internet and service access for VPC-connected functions.

If you use VPC endpoints, you must also configure them to enable private DNS names.

Configure your Amazon VPC security groups with the following rules (at minimum):

- **Inbound rules** – Allow all traffic on the Amazon MSK broker port (9092 for plaintext, 9094 for TLS, 9096 for SASL, 9098 for IAM) for the security groups specified for your event source.
- **Outbound rules** – Allow all traffic on port 443 for all destinations. Allow all traffic on the Amazon MSK broker port (9092 for plaintext, 9094 for TLS, 9096 for SASL, 9098 for IAM) for the security groups specified for your event source.
- If you are using VPC endpoints instead of a NAT gateway, the security groups associated with the VPC endpoints must allow all inbound traffic on port 443 from the event source's security groups.

**Note**

Your Amazon VPC configuration is discoverable through the Amazon MSK API. You don't need to configure it during setup using the create-event-source-mapping command.

For more information about configuring the network, see Setting up AWS Lambda with an Apache Kafka cluster within a VPC on the AWS Compute Blog.
Adding Amazon MSK as an event source

To create an event source mapping, add Amazon MSK as a Lambda function trigger using the Lambda console, an AWS SDK, or the AWS Command Line Interface (AWS CLI). Note that when you add Amazon MSK as a trigger, Lambda assumes the VPC settings of the Amazon MSK cluster, not the Lambda function's VPC settings.

This section describes how to create an event source mapping using the Lambda console and the AWS CLI.

Note

When you update, disable, or delete an event source mapping for Amazon MSK, it can take up to 15 minutes for your changes to take effect. Before this period has elapsed, your event source mapping may continue to process events and invoke your function using your previous settings. This is true even when the status of the event source mapping displayed in the console indicates that your changes have been applied.

Prerequisites

- An Amazon MSK cluster and a Kafka topic. For more information, see Getting Started Using Amazon MSK in the Amazon Managed Streaming for Apache Kafka Developer Guide.
- An execution role with permission to access the AWS resources that your MSK cluster uses.

Customizable consumer group ID

When setting up Kafka as an event source, you can specify a consumer group ID. This consumer group ID is an existing identifier for the Kafka consumer group that you want your Lambda function to join. You can use this feature to seamlessly migrate any ongoing Kafka record processing setups from other consumers to Lambda.

If you specify a consumer group ID and there are other active pollers within that consumer group, Kafka distributes messages across all consumers. In other words, Lambda doesn't receive all message for the Kafka topic. If you want Lambda to handle all messages in the topic, turn off any other pollers in that consumer group.

Additionally, if you specify a consumer group ID, and Kafka finds a valid existing consumer group with the same ID, Lambda ignores the StartingPosition parameter for your event source
mapping. Instead, Lambda begins processing records according to the committed offset of the consumer group. If you specify a consumer group ID, and Kafka cannot find an existing consumer group, then Lambda configures your event source with the specified StartingPosition.

The consumer group ID that you specify must be unique among all your Kafka event sources. After creating a Kafka event source mapping with the consumer group ID specified, you cannot update this value.

On-failure destinations

To retain records of failed invocations or oversized payloads from your Kafka event source, configure an on-failure destination to your function. When an invocation fails, Lambda sends a JSON record containing details of the invocation to your destination.

You can choose between an Amazon SNS topic, Amazon SQS queue, or Amazon S3 bucket as your destination. For SNS topic or SQS queue destinations, Lambda sends the record metadata to the destination. For S3 bucket destinations, Lambda sends the entire invocation record along with the metadata to the destination.

For Lambda to successfully send records to your chosen destination, ensure that your function's execution role contains the relevant permissions. The table also describes how each destination type receives the JSON invocation record.

<table>
<thead>
<tr>
<th>Destination type</th>
<th>Supported for the following event sources</th>
<th>Required permissions</th>
<th>Destination-specific JSON format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon SQS queue</td>
<td>• Kinesis</td>
<td>• sqs:SendMessage</td>
<td>Lambda passes the invocation record metadata as the Message to the destination.</td>
</tr>
<tr>
<td></td>
<td>• DynamoDB</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Self-managed Apache Kafka and Managed Apache Kafka</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon SNS topic</td>
<td>• Kinesis</td>
<td>• sns:Publish</td>
<td>Lambda passes the invocation record metadata as the</td>
</tr>
<tr>
<td></td>
<td>• DynamoDB</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Self-managed Apache Kafka and Managed Apache Kafka</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Destination type | Supported for the following event sources | Required permissions | Destination-specific JSON format
--- | --- | --- | ---
| Managed Apache Kafka | | | Message to the destination.
| Amazon S3 bucket | • Self-managed Apache Kafka and Managed Apache Kafka | • `s3:PutObject`<br>• `s3:ListBuckets` | Lambda stores the invocation record along with its metadata at the destination.

**Tip**

As a best practice, include the minimum permissions required only in your execution role.

---

### SNS and SQS destinations

The following example shows what Lambda sends to an SNS topic or SQS queue destination for a failed Kafka event source invocation. Each of the keys under `recordsInfo` contains both the Kafka topic and partition, separated by a hyphen. For example, for the key "Topic-0", `Topic` is the Kafka topic, and `0` is the partition. For each topic and partition, you can use the offsets and timestamp data to find the original invocation records.

```json
{
  "requestContext": {
    "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
    "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
    "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
    "approximateInvokeCount": 1
  },
  "responseContext": { // null if record is MaximumPayloadSizeExceeded
    "statusCode": 200,
    "executedVersion": "$LATEST",
    "functionError": "Unhandled"
  },
  "version": "1.0",
}```
S3 destinations

For S3 destinations, Lambda sends the entire invocation record along with the metadata to the destination. The following example shows that Lambda sends to an S3 bucket destination for a failed Kafka event source invocation. In addition to all of the fields from the previous example for SQS and SNS destinations, the payload field contains the original invocation record as an escaped JSON string.

```json
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
}
```
Tip

We recommend enabling S3 versioning on your destination bucket.

Configuring on-failure destinations

To configure an on-failure destination using the console, follow these steps:
1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Under Function overview, choose Add destination.
4. For Source, choose Event source mapping invocation.
5. For Event source mapping, choose an event source that's configured for this function.
6. For Condition, select On failure. For event source mapping invocations, this is the only accepted condition.
7. For Destination type, choose the destination type that Lambda sends invocation records to.
8. For Destination, choose a resource.
9. Choose Save.

You can also configure an on-failure destination using the Lambda API. For example, the following CreateEventSourceMapping CLI command adds an SQS on-failure destination to MyFunction:

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-east-1:123456789012:dest-queue"}}'
```

The following UpdateEventSourceMapping CLI command adds an S3 on-failure destination to the Kafka event source associated with the input uuid:

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

To remove a destination, supply an empty string as the argument to the destination-config parameter:

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

**Adding an Amazon MSK trigger (console)**

Follow these steps to add your Amazon MSK cluster and a Kafka topic as a trigger for your Lambda function.
To add an Amazon MSK trigger to your Lambda function (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of your Lambda function.
4. Under Trigger configuration, do the following:
   a. Choose the MSK trigger type.
   b. For MSK cluster, select your cluster.
   c. For Batch size, enter the maximum number of messages to receive in a single batch.
   d. For Batch window, enter the maximum amount of seconds that Lambda spends gathering records before invoking the function.
   e. For Topic name, enter the name of a Kafka topic.
   f. (Optional) For Consumer group ID, enter the ID of a Kafka consumer group to join.
   g. (Optional) For Starting position, choose Latest to start reading the stream from the latest record, Trim horizon to start at the earliest available record, or At timestamp to specify a timestamp to start reading from.
   h. (Optional) For Authentication, choose the secret key for authenticating with the brokers in your MSK cluster.
   i. To create the trigger in a disabled state for testing (recommended), clear Enable trigger. Or, to enable the trigger immediately, select Enable trigger.
5. To create the trigger, choose Add.

Adding an Amazon MSK trigger (AWS CLI)

Use the following example AWS CLI commands to create and view an Amazon MSK trigger for your Lambda function.

Creating a trigger using the AWS CLI

Example — Create event source mapping for cluster that uses IAM authentication

The following example uses the create-event-source-mapping AWS CLI command to map a Lambda function named my-kafka-function to a Kafka topic named AWSKafkaTopic. The topic's starting position is set to LATEST. When the cluster uses IAM role-based authentication, you don't need a SourceAccessConfiguration object. Example:
Example — Create event source mapping for cluster that uses SASL/SCRAM authentication

If the cluster uses SASL/SCRAM authentication, you must include a SourceAccessConfiguration object that specifies SASL_SCRAM_512_AUTH and a Secrets Manager secret ARN.

Example — Create event source mapping for cluster that uses mTLS authentication

If the cluster uses mTLS authentication, you must include a SourceAccessConfiguration object that specifies CLIENT_CERTIFICATE_TLS_AUTH and a Secrets Manager secret ARN.

For more information, see the `CreateEventSourceMapping` API reference documentation.

Viewing the status using the AWS CLI

The following example uses the `get-event-source-mapping` AWS CLI command to describe the status of the event source mapping that you created.
Creating cross-account event source mappings

You can use multi-VPC private connectivity to connect a Lambda function to a provisioned MSK cluster in a different AWS account. Multi-VPC connectivity uses AWS PrivateLink, which keeps all traffic within the AWS network.

Note
You can't create cross-account event source mappings for serverless MSK clusters.

To create a cross-account event source mapping, you must first configure multi-VPC connectivity for the MSK cluster. When you create the event source mapping, use the managed VPC connection ARN instead of the cluster ARN, as shown in the following examples. The CreateEventSourceMapping operation also differs depending on which authentication type the MSK cluster uses.

Example — Create cross-account event source mapping for cluster that uses IAM authentication

When the cluster uses IAM role-based authentication, you don't need a SourceAccessConfiguration object. Example:

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

Example — Create cross-account event source mapping for cluster that uses SASL/SCRAM authentication

If the cluster uses SASL/SCRAM authentication, you must include a SourceAccessConfiguration object that specifies SASL_SCRAM_512_AUTH and a Secrets Manager secret ARN.

There are two ways to use secrets for cross-account Amazon MSK event source mappings with SASL/SCRAM authentication:
• Create a secret in the Lambda function account and sync it with the cluster secret. Create a rotation to keep the two secrets in sync. This option allows you to control the secret from the function account.

• Use the secret that's associated with the MSK cluster. This secret must allow cross-account access to the Lambda function account. For more information, see Permissions to AWS Secrets Manager secrets for users in a different account.

```bash
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[["Type": "SASL_SCRAN_512_AUTH","URI": "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"]]'
```

Example — Create cross-account event source mapping for cluster that uses mTLS authentication

If the cluster uses mTLS authentication, you must include a SourceAccessConfiguration object that specifies CLIENT_CERTIFICATE_TLS_AUTH and a Secrets Manager secret ARN. The secret can be stored in the cluster account or the Lambda function account.

```bash
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[["Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI": "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"]]'
```

Auto scaling of the Amazon MSK event source

When you initially create an Amazon MSK event source, Lambda allocates one consumer to process all partitions in the Kafka topic. Each consumer has multiple processors running in parallel to handle increased workloads. Additionally, Lambda automatically scales up or down the number of consumers, based on workload. To preserve message ordering in each partition, the maximum number of consumers is one consumer per partition in the topic.
In one-minute intervals, Lambda evaluates the consumer offset lag of all the partitions in the topic. If the lag is too high, the partition is receiving messages faster than Lambda can process them. If necessary, Lambda adds or removes consumers from the topic. The scaling process of adding or removing consumers occurs within three minutes of evaluation.

If your target Lambda function is throttled, Lambda reduces the number of consumers. This action reduces the workload on the function by reducing the number of messages that consumers can retrieve and send to the function.

To monitor the throughput of your Kafka topic, view the Offset lag metric Lambda emits while your function processes records.

To check how many function invocations occur in parallel, you can also monitor the concurrency metrics for your function.

**Polling and stream starting positions**

Be aware that stream polling during event source mapping creation and updates is eventually consistent.

- During event source mapping creation, it may take several minutes to start polling events from the stream.
- During event source mapping updates, it may take several minutes to stop and restart polling events from the stream.

This behavior means that if you specify LATEST as the starting position for the stream, the event source mapping could miss events during creation or updates. To ensure that no events are missed, specify the stream starting position as TRIM_HORIZON or AT_TIMESTAMP.

**Amazon CloudWatch metrics**

Lambda emits the OffsetLag metric while your function processes records. The value of this metric is the difference in offset between the last record written to the Kafka event source topic and the last record that your function's consumer group processed. You can use OffsetLag to estimate the latency between when a record is added and when your consumer group processes it.

An increasing trend in OffsetLag can indicate issues with pollers in your function's consumer group. For more information, see Working with Lambda function metrics.
Amazon MSK configuration parameters

All Lambda event source types share the same CreateEventSourceMapping and UpdateEventSourceMapping API operations. However, only some of the parameters apply to Amazon MSK.

Event source parameters that apply to Amazon MSK

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Default</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AmazonManagedKafkaEventSourceConfig</td>
<td>N</td>
<td>Contains the ConsumerGroupId field, which defaults to a unique value.</td>
<td>Can set only on Create</td>
</tr>
<tr>
<td>BatchSize</td>
<td>N</td>
<td>100</td>
<td>Maximum: 10,000</td>
</tr>
<tr>
<td>Enabled</td>
<td>N</td>
<td>Enabled</td>
<td></td>
</tr>
<tr>
<td>EventSourceArn</td>
<td>Y</td>
<td></td>
<td>Can set only on Create</td>
</tr>
<tr>
<td>FunctionName</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FilterCriteria</td>
<td>N</td>
<td></td>
<td>Lambda event filtering</td>
</tr>
<tr>
<td>MaximumBatchingWindowInSeconds</td>
<td>N</td>
<td>500 ms</td>
<td>Batching behavior</td>
</tr>
<tr>
<td>SourceAccessConfigurations</td>
<td>N</td>
<td>No credentials</td>
<td>SASL/SCRAM or CLIENT_CERTIFICATE_TLS_AUTH (MutualTLS) authentication credentials for your event source</td>
</tr>
<tr>
<td>Parameter</td>
<td>Required</td>
<td>Default</td>
<td>Notes</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
<td>---------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>StartingPosition</td>
<td>Y</td>
<td></td>
<td>AT_TIMESTAMP, TRIM_HORIZON, or LATEST</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Can set only on Create</td>
</tr>
<tr>
<td>StartingPositionTimestamp</td>
<td>N</td>
<td></td>
<td>Required if StartingPosition is set to AT_TIMESTAMP</td>
</tr>
<tr>
<td>Topics</td>
<td>Y</td>
<td></td>
<td>Kafka topic name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Can set only on Create</td>
</tr>
</tbody>
</table>
Using AWS Lambda with Amazon RDS

You can connect a Lambda function to an Amazon Relational Database Service (Amazon RDS) database directly and through an Amazon RDS Proxy. Direct connections are useful in simple scenarios, and proxies are recommended for production. A database proxy manages a pool of shared database connections which enables your function to reach high concurrency levels without exhausting database connections.

We recommend using Amazon RDS Proxy for Lambda functions that make frequent short database connections, or open and close large numbers of database connections.

Configuring your function

In the Lambda console, you can configure and even create certain database instances and proxy resources. See RDS databases under configuration. Alternatively, you can also create and configure connections to Lambda functions in the Amazon RDS console.

• To connect to a database, your function must be in the same Amazon VPC where your database runs.
• You can use Amazon RDS databases with MySQL, MariaDB, PostgreSQL, or Microsoft SQL Server engines.
• You can also use Aurora DB clusters with MySQL or PostgreSQL engines.
• You need to provide a Secrets Manager secret for database authentication.
• An IAM role must provide permission to use the secret, and a trust policy must allow Amazon RDS to assume the role.

Amazon RDS charges an hourly rate for proxies based on the database instance size, see RDS Proxy pricing for details. For more information on proxy connections in general, see Using Amazon RDS Proxy in the Amazon RDS User Guide.

Lambda and Amazon RDS setup

Both Lambda and Amazon RDS consoles will assist you in automatically configuring some of the required resources to make a connection between Lambda and Amazon RDS.
You can use Lambda to process event notifications from an Amazon RDS database. Amazon RDS sends notifications to an Amazon Simple Notification Service (Amazon SNS) topic, which you can configure to invoke a Lambda function. Amazon SNS wraps the message from Amazon RDS in its own event document and sends it to your function.

For more information about configuring an Amazon RDS database to send notifications, see [Using Amazon RDS event notifications](#).

### Example Amazon RDS message in an Amazon SNS event

```
{
    "Records": [
        {
            "EventVersion": "1.0",
            "EventSource": "aws:sns",
            "Sns": {
                "SignatureVersion": "1",
                "Timestamp": "2023-01-02T12:45:07.000Z",
                "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEka6RibDsvpi+tE/1+82j...65r==",
                "SigningCertUrl": "https://sns.us-east-2.amazonaws.com/SimpleNotificationService-ac565b8b1a6c5d002d285f9598a1d9b.pem",
                "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
                "MessageAttributes": {},
                "Type": "Notification",
                "Subject": "RDS Notification Message"
            }
        }
    ]
}
```
Lambda and Amazon RDS tutorial

- **Using a Lambda function to access an Amazon RDS database** – From the Amazon RDS User Guide, learn how to use a Lambda function to write data to an Amazon RDS database through an Amazon RDS Proxy. Your Lambda function will read records from an Amazon SQS queue and write new items to a table in your database whenever a message is added.
Using AWS Lambda with Amazon S3

You can use Lambda to process event notifications from Amazon Simple Storage Service. Amazon S3 can send an event to a Lambda function when an object is created or deleted. You configure notification settings on a bucket, and grant Amazon S3 permission to invoke a function on the function's resource-based permissions policy.

⚠️ Warning

If your Lambda function uses the same bucket that triggers it, it could cause the function to run in a loop. For example, if the bucket triggers a function each time an object is uploaded, and the function uploads an object to the bucket, then the function indirectly triggers itself. To avoid this, use two buckets, or configure the trigger to only apply to a prefix used for incoming objects.

Amazon S3 invokes your function asynchronously with an event that contains details about the object. The following example shows an event that Amazon S3 sent when a deployment package was uploaded to Amazon S3.

**Example Amazon S3 notification event**

```json
{
    "Records": [
        {
            "eventVersion": "2.1",
            "eventSource": "aws:s3",
            "awsRegion": "us-east-2",
            "eventTime": "2019-09-03T19:37:27.192Z",
            "eventName": "ObjectCreated:Put",
            "userIdentity": {
                "principalId": "AWS:AIDAINPONIXQXHT3IKHL2"
            },
            "requestParameters": {
                "sourceIPAddress": "205.255.255.255"
            },
            "responseElements": {
                "x-amz-request-id": "D82B88E5F771F645",
                "x-amz-id-2": "vlR7PnpV2Ce811oPRw6j1Upck7Jo5ZsQjryTjk1c5aLWGVPZLj5NeC6qMa0emYBDX0o6QBU0Wo=
            }
        }
    ]
}
```
To invoke your function, Amazon S3 needs permission from the function's resource-based policy. When you configure an Amazon S3 trigger in the Lambda console, the console modifies the resource-based policy to allow Amazon S3 to invoke the function if the bucket name and account ID match. If you configure the notification in Amazon S3, you use the Lambda API to update the policy. You can also use the Lambda API to grant permission to another account, or restrict permission to a designated alias.

If your function uses the AWS SDK to manage Amazon S3 resources, it also needs Amazon S3 permissions in its execution role.

Topics

- Tutorial: Using an Amazon S3 trigger to invoke a Lambda function
- Tutorial: Using an Amazon S3 trigger to create thumbnail images

**Tutorial: Using an Amazon S3 trigger to invoke a Lambda function**

In this tutorial, you use the console to create a Lambda function and configure a trigger for an Amazon Simple Storage Service (Amazon S3) bucket. Every time that you add an object to your Amazon S3 bucket, your function runs and outputs the object type to Amazon CloudWatch Logs.
You can use a Lambda function with an Amazon S3 trigger to perform many types of file processing tasks. For example, you can use a Lambda function to create a thumbnail whenever an image file is uploaded to your Amazon S3 bucket, or to convert uploaded documents into different formats. After you've completed this tutorial, you can try the Using an Amazon S3 trigger to create thumbnail images tutorial to perform an image processing task.

To complete this tutorial, you carry out the following steps:

1. Create an Amazon S3 bucket.
2. Create a Lambda function that returns the object type of objects in an Amazon S3 bucket.
3. Configure a Lambda trigger that invokes your function when objects are uploaded to your bucket.
4. Test your function, first with a dummy event, and then using the trigger.

By completing these steps, you’ll learn how to configure a Lambda function to run whenever objects are added to or deleted from an Amazon S3 bucket. You can complete this tutorial using only the AWS Management Console.
Prerequisites

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

When you sign up for an AWS account, an AWS account root user is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to an administrative user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing My Account.

Create an administrative user

After you sign up for an AWS account, create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing Root user and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the AWS Sign-In User Guide.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the IAM User Guide.
Create an administrative user

- For your daily administrative tasks, grant administrative access to an administrative user in AWS IAM Identity Center.

  For instructions, see [Getting started](#) in the *AWS IAM Identity Center User Guide*.

Sign in as the administrative user

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see [Signing in to the AWS access portal](#) in the *AWS Sign-In User Guide*.

Create an Amazon S3 bucket

First create an Amazon S3 bucket using the AWS Management Console.

**To create an Amazon S3 bucket**

1. Open the [Amazon S3 console](#) and select the **Buckets** page.
2. Choose **Create bucket**.
3. Under **General configuration**, do the following:
   a. For **Bucket name**, enter a globally unique name that meets the Amazon S3 **Bucket naming rules**. Bucket names can contain only lower case letters, numbers, dots (.), and hyphens (-).
b. For **AWS Region**, choose a Region. Later in the tutorial, you must create your Lambda function in the same Region.

4. Leave all other options set to their default values and choose **Create bucket**.

**Upload a test object to your bucket**

![Diagram of S3 Bucket, Function, Trigger, and Test]

Later in the tutorial, you'll test your Lambda function in the Lambda console. To confirm that your function's code is working correctly, your Amazon S3 bucket needs to contain a test object. This object can be any file you choose (for example HappyFace.jpg).

**To upload a test object**

1. Open the **Buckets** page of the Amazon S3 console and choose the bucket you created during the previous step.
2. Choose **Upload**.
3. Choose **Add files** and use the file selector to choose the object you want to upload.
4. Choose **Open**, then choose **Upload**.

When you test your function code later in the tutorial, you pass it data containing the file name of the object you uploaded, so make a note of it now.
Create a permissions policy

Before you can create an execution role for your Lambda function, you first create a permissions policy to give your function permission to access the required AWS resources. For this tutorial, the policy allows Lambda to get objects from an Amazon S3 bucket and to write to Amazon CloudWatch Logs.

To create the policy

1. Open the Policies page of the IAM console.
2. Choose Create Policy.
3. Choose the JSON tab, and then paste the following custom policy into the JSON editor.

```json
{
"Version": "2012-10-17",
"Statement": [
{
"Effect": "Allow",
"Action": [
"logs:PutLogEvents",
"logs:CreateLogGroup",
"logs:CreateLogStream"
],
"Resource": "arn:aws:logs:*:*:*
},
{
"Effect": "Allow",
"Action": [
"s3:GetObject"
]
}
}
```
4. Choose **Next: Tags**.
5. Choose **Next: Review**.
7. Choose **Create policy**.

### Create an execution role

![Diagram of S3 Bucket, Function, Trigger, and Test procedures]

An **execution role** is an AWS Identity and Access Management (IAM) role that grants a Lambda function permission to access AWS services and resources. To enable your function to get objects from an Amazon S3 bucket, you attach the permissions policy you created in the previous step.

**To create an execution role and attach your custom permissions policy**

1. Open the **Roles page** of the IAM console.
2. Choose **Create role**.
3. For the type of trusted entity, choose **AWS service**, then for the use case, choose **Lambda**.
4. Choose **Next**.
5. In the policy search box, enter `s3-trigger-tutorial`.
6. In the search results, select the policy that you created (`s3-trigger-tutorial`), and then choose **Next**.
7. Under **Role details**, for the **Role name**, enter `lambda-s3-trigger-role`, then choose **Create role**.
Create the Lambda function

In this example, you create a Lambda function in the console using the Node.js 16.x runtime. The function you create in the console contains some basic ‘Hello World’ code. In the next step, you'll replace this with the function code to get an object from your Amazon S3 bucket.

To create the Lambda function

1. Open the Functions page of the Lambda console.
2. Make sure you're working in the same AWS Region you created your Amazon S3 bucket in. You can change your Region using the drop-down list at the top of the screen.

3. Choose Create function.
4. Choose Author from scratch
5. Under Basic information, do the following:
   a. For Function name, enter s3-trigger-tutorial
   b. For Runtime, choose Node.js 16.x.
   c. For Architecture, choose x86_64.
6. In the **Change default execution role** tab, do the following:

   a. Expand the tab, then choose **Use an existing role**.
   
   b. Select the `lambda-s3-trigger-role` you created earlier.

7. Choose **Create function**.

**Deploy the function code**

![Diagram of deploying a function](image)

Your Lambda function will retrieve the key name of the uploaded object and the name of the bucket from the event parameter it receives from Amazon S3. The function then uses the `HeadObject` API call in the AWS SDK for JavaScript to get the object type for the uploaded object.

This tutorial uses the Node.js 16.x runtime, but we've also provided example code files for other runtimes. You can select the tab in the following box to see the code for the runtime you're interested in. The JavaScript code you'll deploy is the first example shown in the tab labeled **JavaScript**.

**.NET**

**AWS SDK for .NET**

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming an S3 event with Lambda using .NET.
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.

namespace S3Integration
{
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }

        public async Task<string> Handler(S3Event evt, ILambdaContext context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
                var key = HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");
                var objectResult = await _s3Client.GetObjectAsync(bucket, key);
            }
        }
    }
}
Go

SDK for Go V2

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Go.

```go
package main

import (
    "context"
    "log"

    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/s3"
)

func handler(ctx context.Context, s3Event events.S3Event) error {
    sdkConfig, err := config.LoadDefaultConfig(ctx)
    if err != nil {
        context.Logger.LogLine("Error processing request - 
        {e.Message}");
        return string.Empty;
    }
    return objectResult.Key;
}
```
```
log.Printf("failed to load default config: %s", err)
return err
}
s3Client := s3.NewFromConfig(sdkConfig)

for _, record := range s3Event.Records {
    key := record.S3.Object.URLDecodedKey
    headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
        Bucket: &bucket,
        Key:    &key,
    })
    if err != nil {
        log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
        return err
    }
    log.Printf("successfully retrieved %s/%s of type %s", bucket, key, *headOutput.ContentType)
}
return nil
}

func main() {
    lambda.Start(handler)
}
```

Java

**SDK for Java 2.x**

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming an S3 event with Lambda using Java.

```java
package example;
```
import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNotificationRecord;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger = LoggerFactory.getLogger(Handler.class);

    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
            S3EventNotificationRecord record = s3event.getRecords().get(0);
            String srcBucket = record.getS3().getBucket().getName();
            String srcKey = record.getS3().getObject().getUrlDecodedKey();

            S3Client s3Client = S3Client.builder().build();
            HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
            srcKey);

            logger.info("Successfully retrieved " + srcBucket + "/" + srcKey + " of type " + headObject.contentType());

            return "Ok";
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String bucket, String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
            .bucket(bucket)
            .key(key)
            .build();
        return s3Client.headObject(headObjectRequest);
    }
}
JavaScript

**SDK for JavaScript (v2)**

> Note
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

### Consuming an S3 event with Lambda using JavaScript.

```javascript
const aws = require('aws-sdk');

const s3 = new aws.S3({ apiVersion: '2006-03-01' });

exports.handler = async (event, context) => {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, ' '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.headObject(params).promise();
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make sure they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```
Consuming an S3 event with Lambda using TypeScript.

```typescript
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> => {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, ' '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make sure they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

Python

**SDK for Python (Boto3)**

- **Note**
  There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Python.
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')

def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'], encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they exist and your bucket is in the same region as this function.'.format(key, bucket))
        raise e

Rust

SDK for Rust

ℹ️ Note

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Rust.

use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Main function
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
    tracing::info!(records = ?evt.payload.records.len(), "Received request from SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket = evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket name to exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object key to exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
To deploy the function code

1. Open the Functions page of the Lambda console.
2. Choose the function you created in the previous step (s3-trigger-tutorial).
3. Choose the Code tab.
4. Copy and paste the provided JavaScript code into the index.js tab in the Code source pane.
5. Choose Deploy.

Create the Amazon S3 trigger

Now you’ve deployed your function code, you create the Amazon S3 trigger that will invoke your function.
To create the Amazon S3 trigger

1. In the **Function overview** pane of your function’s console page, choose **Add trigger**.
2. Select **S3**.
3. Under **Bucket**, select the bucket you created earlier in the tutorial.
4. Under **Event types**, select **All object create events**. You can also configure a trigger to invoke Lambda when an object is deleted, but we won’t be using that option in this tutorial.
5. Under **Recursive invocation**, select the check box to acknowledge that using the same Amazon S3 bucket for input and output is not recommended. You can learn more about recursive invocation patterns in Lambda by reading [Recursive patterns that cause run-away Lambda functions](#) in Serverless Land.
6. Choose **Add**.

Test your Lambda function with a dummy event

Now that you’ve created and configured your Lambda function, you’re ready to test it. You first test your function by sending it a dummy Amazon S3 event to confirm it’s working correctly.

To test the Lambda function with a dummy event

1. In the Lambda console page for your function, choose the **Code** tab.
2. In the **Code source** pane, choose **Test**.
3. In the **Configure test event** box, do the following:
   a. For **Event name**, enter MyTestEvent.
   b. For **Template**, choose **S3 Put**.
c. In the Event JSON, replace the following values:

- Replace `us-east-1` with the region you created your Amazon S3 bucket in.
- Replace both instances of `my-bucket` with the name of your own Amazon S3 bucket.
- Replace `test%2Fkey` with the name of the test object you uploaded to your bucket earlier (for example, `HappyFace.jpg`).

```json
{
    "Records": [
        {
            "eventVersion": "2.0",
            "eventSource": "aws:s3",
            "awsRegion": "us-east-1",
            "eventTime": "1970-01-01T00:00:00.000Z",
            "eventName": "ObjectCreated:Put",
            "userIdentity": {
                "principalId": "EXAMPLE"
            },
            "requestParameters": {
                "sourceIPAddress": "127.0.0.1"
            },
            "responseElements": {
                "x-amz-request-id": "EXAMPLE123456789",
                "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/mnopqrstuvwxyzABCDEFGH"
            },
            "s3": {
                "s3SchemaVersion": "1.0",
                "configurationId": "testConfigRule",
                "bucket": {
                    "name": "my-bucket",
                    "ownerIdentity": {
                        "principalId": "EXAMPLE"
                    },
                    "arn": "arn:aws:s3:::my-bucket"
                },
                "object": {
                    "key": "test%2Fkey",
                    "size": 1024,
                    "eTag": "0123456789abcdef0123456789abcdef",
                    "sequencer": "0A1B2C3D4E5F678901"
                }
            }
        }
    ]
}
```
d. Choose Save.

4. In the **Code source** pane, choose **Test**.

5. If your function runs successfully, you’ll see output similar to the following in the **Execution results** tab.

```
Response
"image/jpeg"

Function Logs
START RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6 Version: $LATEST
2021-02-18T21:40:59.280Z  12b3cae7-5f4e-415e-93e6-416b8f8b66e6 INFO INPUT
BUCKET AND KEY:  { Bucket: 'my-bucket', Key: 'HappyFace.jpg' }
2021-02-18T21:41:00.215Z  12b3cae7-5f4e-415e-93e6-416b8f8b66e6 INFO CONTENT
   TYPE: image/jpeg
END RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6
REPORT RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6
   Duration: 976.25 ms
   Billed Duration: 977 ms
   Memory Size: 128 MB
   Max Memory Used: 90 MB
   Init Duration: 430.47 ms

Request ID
12b3cae7-5f4e-415e-93e6-416b8f8b66e6
```

**Test the Lambda function with the Amazon S3 trigger**

**S3 Bucket**
- Create bucket
- Upload object

**Function**
- Create policy
- Create role
- Create function
- Deploy code

**Trigger**
- Configure trigger

**Test**
- Test with dummy event
- Test with trigger

---

*Tutorial: Use an S3 trigger*
To test your function with the configured trigger, you upload an object to your Amazon S3 bucket using the console. To verify that your Lambda function has been invoked correctly, you then use CloudWatch Logs to view your function’s output.

**To upload an object to your Amazon S3 bucket**

1. Open the **Buckets** page of the Amazon S3 console and choose the bucket you created earlier.
2. Choose **Upload**.
3. Choose **Add files** and use the file selector to choose an object you want to upload. This object can be any file you choose.
4. Choose **Open**, then choose **Upload**.

**To verify correct operation using CloudWatch Logs**

1. Open the **CloudWatch** console.
2. Make sure you're working in the same AWS Region you created your Lambda function in. You can change your Region using the drop-down list at the top of the screen.
3. Choose **Logs**, then choose **Log groups**.
4. Choose the log group for your function (**/aws/lambda/s3-trigger-tutorial**).
5. Under **Log streams**, choose the most recent log stream.
6. If your function has been invoked correctly in response to your Amazon S3 trigger, you’ll see output similar to the following. The CONTENT TYPE you see depends on the type of file you uploaded to your bucket.

```plaintext
2022-05-09T23:17:28.702Z 0cae7f5a-b0af-4c73-8563-a3430333cc10 INFO CONTENT TYPE: image/jpeg
```
Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose Actions, Delete.
4. Type delete in the text input field and choose Delete.

To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose Delete.
4. Enter the name of the role in the text input field and choose Delete.

To delete the S3 bucket

1. Open the Amazon S3 console.
2. Select the bucket you created.
3. Choose Delete.
4. Enter the name of the bucket in the text input field.
5. Choose Delete bucket.

Next steps

Try the more advanced tutorial. In this tutorial, the Amazon S3 trigger invokes a function to create a thumbnail image for each image file that is uploaded to your bucket. This tutorial requires a moderate level of AWS and Lambda domain knowledge. You use the AWS Command Line Interface (AWS CLI) to create resources, and you create a .zip file archive deployment package for your function and its dependencies.
Tutorial: Using an Amazon S3 trigger to create thumbnail images

In this tutorial, you create and configure a Lambda function that resizes images added to an Amazon Simple Storage Service (Amazon S3) bucket. When you add an image file to your bucket, Amazon S3 invokes your Lambda function. The function then creates a thumbnail version of the image and outputs it to a different Amazon S3 bucket.

To complete this tutorial, you carry out the following steps:

1. Create source and destination Amazon S3 buckets and upload a sample image.
2. Create a Lambda function that resizes an image and outputs a thumbnail to an Amazon S3 bucket.
3. Configure a Lambda trigger that invokes your function when objects are uploaded to your source bucket.
4. Test your function, first with a dummy event, and then by uploading an image to your source bucket.

By completing these steps, you’ll learn how to use Lambda to carry out a file processing task on objects added to an Amazon S3 bucket. You can complete this tutorial using the AWS Command Line Interface (AWS CLI) or the AWS Management Console.

If you're looking for a simpler example to learn how to configure an Amazon S3 trigger for Lambda, you can try Tutorial: Using an Amazon S3 trigger to invoke a Lambda function.

Topics
- Prerequisites
- Create two Amazon S3 buckets
- Upload a test image to your source bucket
- Create a permissions policy
- Create an execution role
- Create the function deployment package
- Create the Lambda function
- Configure Amazon S3 to invoke the function
- Test your Lambda function with a dummy event
- Test your function using the Amazon S3 trigger
- Clean up your resources

**Prerequisites**

**Sign up for an AWS account**

If you do not have an AWS account, complete the following steps to create one.

**To sign up for an AWS account**

2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to an administrative user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to [https://aws.amazon.com/](https://aws.amazon.com/) and choosing My Account.
Create an administrative user

After you sign up for an AWS account, create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing Root user and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the AWS Sign-In User Guide.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the IAM User Guide.

Create an administrative user

• For your daily administrative tasks, grant administrative access to an administrative user in AWS IAM Identity Center.

   For instructions, see Getting started in the AWS IAM Identity Center User Guide.

Sign in as the administrative user

• To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

   For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the AWS Sign-In User Guide.

If you want to use the AWS CLI to complete the tutorial, install the latest version of the AWS Command Line Interface.

For your Lambda function code, you can use Python, Node.js or Java. Install the language support tools and a package manager for the language that you want to use. For Java, the example commands to create your deployment package use the Apache Maven build automation tool.
Create two Amazon S3 buckets

First create two Amazon S3 buckets. The first bucket is the source bucket you will upload your images to. The second bucket is used by Lambda to save the resized thumbnail when you invoke your function.

AWS Management Console

To create the Amazon S3 buckets (console)

1. Open the Buckets page of the Amazon S3 console.
2. Choose Create bucket.
3. Under General configuration, do the following:
   a. For Bucket name, enter a globally unique name that meets the Amazon S3 Bucket naming rules. Bucket names can contain only lower case letters, numbers, dots (.), and hyphens (-).
   b. For AWS Region, choose the AWS Region closest to your geographical location. Later in the tutorial, you must create your Lambda function in the same AWS Region, so make a note of the region you chose.
4. Leave all other options set to their default values and choose Create bucket.
5. Repeat steps 1 to 4 to create your destination bucket. For Bucket name, enter SOURCEBUCKET-resized, where SOURCEBUCKET is the name of the source bucket you just created.
AWS CLI

To create the Amazon S3 buckets (AWS CLI)

1. Run the following CLI command to create your source bucket. The name you choose for your bucket must be globally unique and follow the Amazon S3 Bucket naming rules. Names can only contain lower case letters, numbers, dots (.), and hyphens (-). For region and LocationConstraint, choose the AWS Region closest to your geographical location.

   ```bash
   aws s3api create-bucket --bucket SOURCEBUCKET --region us-west-2 \\   --create-bucket-configuration LocationConstraint=us-west-2
   ```

   Later in the tutorial, you must create your Lambda function in the same AWS Region as your source bucket, so make a note of the region you chose.

2. Run the following command to create your destination bucket. For the bucket name, you must use `SOURCEBUCKET-resized`, where `SOURCEBUCKET` is the name of the source bucket you created in step 1. For region and LocationConstraint, choose the same AWS Region you used to create your source bucket.

   ```bash
   aws s3api create-bucket --bucket SOURCEBUCKET-resized --region us-west-2 \\   --create-bucket-configuration LocationConstraint=us-west-2
   ```

Upload a test image to your source bucket

Later in the tutorial, you’ll test your Lambda function by invoking it using the AWS CLI or the Lambda console. To confirm that your function is operating correctly, your source bucket needs to contain a test image. This image can be any JPG or PNG file you choose.
AWS Management Console

To upload a test image to your source bucket (console)

1. Open the **Buckets** page of the Amazon S3 console.
2. Choose **Upload**.
3. Choose **Add file** and use the file selector to choose the object you want to upload.
4. Choose **Open**, then choose **Upload**.

AWS CLI

To upload a test image to your source bucket (AWS CLI)

- From the directory containing the image you want to upload, run the following CLI command. Replace the **--bucket** parameter with the name of your source bucket. For the **--key** and **--body** parameters, use the filename of your test image.

```
aws s3api put-object --bucket SOURCEBUCKET --key HappyFace.jpg --body ./HappyFace.jpg
```

Create a permissions policy

The first step in creating your Lambda function is to create a permissions policy. This policy gives your function the permissions it needs to access other AWS resources. For this tutorial, the policy gives Lambda read and write permissions for Amazon S3 buckets and allows it to write to Amazon CloudWatch Logs logs.
AWS Management Console

To create the policy (console)

1. Open the Policies page of the AWS Identity and Access Management (IAM) console.
2. Choose Create policy.
3. Choose the JSON tab, and then paste the following custom policy into the JSON editor.

```json
{
   "Version": "2012-10-17",
   "Statement": [
     {
       "Effect": "Allow",
       "Action": [
         "logs:PutLogEvents",
         "logs:CreateLogGroup",
         "logs:CreateLogStream"
       ],
       "Resource": "arn:aws:logs:*:*:*"
     },
     {
       "Effect": "Allow",
       "Action": [
         "s3:GetObject"
       ],
       "Resource": "arn:aws:s3:::*/*"
     },
     {
       "Effect": "Allow",
       "Action": [
         "s3:PutObject"
       ],
       "Resource": "arn:aws:s3:::*/*"
     }
   ]
}
```
4. Choose Next: Tags.
5. Choose Next: Review.
6. Under Review policy, for Name, enter LambdaS3Policy.
7. Choose Create policy.
AWS CLI

To create the policy (AWS CLI)

1. Save the following JSON in a file named policy.json.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

2. From the directory you saved the JSON policy document in, run the following CLI command.

```
aws iam create-policy --policy-name LambdaS3Policy --policy-document file://policy.json
```
Create an execution role

An execution role is an IAM role that grants a Lambda function permission to access AWS services and resources. To give your function read and write access to an Amazon S3 bucket, you attach the permissions policy you created in the previous step.

AWS Management Console

To create an execution role and attach your permissions policy (console)

1. Open the [Roles](#) page of the (IAM) console.
2. Choose [Create role](#).
3. For **Trusted entity type**, select **AWS service**, and for **Use case**, select **Lambda**.
4. Choose [Next](#).
5. Add the permissions policy you created in the previous step by doing the following:
   a. In the policy search box, enter **LambdaS3Policy**.
   b. In the search results, select the check box for LambdaS3Policy.
   c. Choose [Next](#).
6. Under **Role details**, for the **Role name** enter **LambdaS3Role**.
7. Choose [Create role](#).
To create an execution role and attach your permissions policy (AWS CLI)

1. Save the following JSON in a file named trust-policy.json. This trust policy allows Lambda to use the role's permissions by giving the service principal lambda.amazonaws.com permission to call the AWS Security Token Service (AWS STS) AssumeRole action.

   ```json
   {
     "Version": "2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Principal": {
           "Service": "lambda.amazonaws.com"
         },
         "Action": "sts:AssumeRole"
       }
     ]
   }```

2. From the directory you saved the JSON trust policy document in, run the following CLI command to create the execution role.

   ```bash
   aws iam create-role --role-name LambdaS3Role --assume-role-policy-document file://trust-policy.json
   ```

3. To attach the permissions policy you created in the previous step, run the following CLI command. Replace the AWS account number in the policy’s ARN with your own account number.

   ```bash
   aws iam attach-role-policy --role-name LambdaS3Role --policy-arn arn:aws:iam::123456789012:policy/LambdaS3Policy
   ```
Create the function deployment package

To create your function, you create a deployment package containing your function code and its dependencies. For this CreateThumbnail function, your function code uses a separate library for the image resizing. Follow the instructions for your chosen language to create a deployment package containing the required library.

Node.js

To create the deployment package (Node.js)

1. Create a directory named `lambda-s3` for your function code and dependencies and navigate into it.

   ```bash
   mkdir lambda-s3
   cd lambda-s3
   ```

2. Save the following function code in a file named `index.mjs`.

   ```javascript
   // dependencies
   import { S3Client, GetObjectCommand, PutObjectCommand } from '@aws-sdk/client-s3';
   import { Readable } from 'stream';
   import sharp from 'sharp';
   import util from 'util';

   // create S3 client
   const s3 = new S3Client({region: 'us-west-2'});
   ```
// define the handler function
export const handler = async (event, context) => {

// Read options from the event parameter and get the source bucket
console.log("Reading options from event:\n", util.inspect(event, {depth: 5}));
const srcBucket = event.Records[0].s3.bucket.name;

// Object key may have spaces or unicode non-ASCII characters
const srcKey = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, " ")рус);
const dstBucket = srcBucket + "-resized";
const dstKey = "resized-" + srcKey;

// Infer the image type from the file suffix
const typeMatch = srcKey.match(/\.(\[^.]*$/);  
if (!typeMatch) {
  console.log("Could not determine the image type.");
  return;
}

// Check that the image type is supported
const imageType = typeMatch[1].toLowerCase();
if (imageType != "jpg" && imageType != "png") {
  console.log("Unsupported image type: ${imageType}");
  return;
}

// Get the image from the source bucket. GetObjectCommand returns a stream.
try {
  const params = {
    Bucket: srcBucket,
    Key: srcKey
  };
  var response = await s3.send(new GetObjectCommand(params));
  var stream = response.Body;

  // Convert stream to buffer to pass to sharp resize function.
  if (stream instanceof Readable) {
    var content_buffer = Buffer.concat(await stream.toArray());
  } else {
    throw new Error('Unknown object stream type');
  }
}
3. In your lambda-s3 directory, install the sharp library using npm.
npm install sharp

The npm install command creates a node_modules directory for your modules. After this step, your directory structure should look like the following.

```
lambda-s3
 | - index.mjs
 | - node_modules
 |   | - base64js
 |   | - bl
 |   | - buffer
 ...  
 | - package-lock.json
 | - package.json
```

4. Create a .zip deployment package containing your function code and its dependencies. In MacOS and Linux, run the following command.

```
zip -r function.zip .
```

In Windows, use your preferred zip utility to create a .zip file. Ensure that your index.mjs, package.json, and package-lock.json files and your node_modules directory are all at the root of your .zip file.

**Python**

**To create the deployment package (Python)**

1. Save the example code as a file named lambda_function.py.

```python
import boto3
import os
import sys
import uuid
from urllib.parse import unquote_plus
from PIL import Image
import PIL.Image

s3_client = boto3.client('s3')
```
def resize_image(image_path, resized_path):
    with Image.open(image_path) as image:
        image.thumbnail(tuple(x / 2 for x in image.size))
        image.save(resized_path)

def lambda_handler(event, context):
    for record in event['Records']:
        bucket = record['s3']['bucket']['name']
        key = unquote_plus(record['s3']['object']['key'])
        tmpkey = key.replace('/', '')
        download_path = '/tmp/{}{}'.format(uuid.uuid4(), tmpkey)
        upload_path = '/tmp/resized-{}'.format(tmpkey)
        s3_client.download_file(bucket, key, download_path)
        resize_image(download_path, upload_path)
        s3_client.upload_file(upload_path, '{}-resized'.format(bucket), 'resized-{}'.format(key))

2. In the same directory in which you created your `lambda_function.py` file, create a new
directory named `package` and install the Pillow (PIL) library and the AWS SDK for Python
(Boto3). Although the Lambda Python runtime includes a version of the Boto3 SDK, we
recommend that you add all of your function's dependencies to your deployment package,
even if they are included in the runtime. For more information, see Runtime dependencies
in Python.

    mkdir package
    pip install \  
    --platform manylinux2014_x86_64 \  
    --target=package \  
    --implementation cp \  
    --python-version 3.9 \  
    --only-binary=:all: --upgrade \  
    pillow boto3

The Pillow library contains C/C++ code. By using the --platform
manylinux_2014_x86_64 and --only-binary=:all: options, pip will download and
install a version of Pillow that contains pre-compiled binaries compatible with the Amazon
Linux 2 operating system. This ensures that your deployment package will work in the
Lambda execution environment, regardless of the operating system and architecture of
your local build machine.
3. Create a .zip file containing your application code and the Pillow and Boto3 libraries. In Linux or MacOS, run the following commands from your command line interface.

```bash
cd package
zip -r ../lambda_function.zip .
cd..
zip lambda_function.zip lambda_function.py
```

In Windows, use your preferred zip tool to create the `lambda_function.zip` file. Make sure that your `lambda_function.py` file and the folders containing your dependencies are all at the root of the .zip file.

You can also create your deployment package using a Python virtual environment. See [Working with .zip file archives for Python Lambda functions](#).

### Java

**Prerequisites**

Make sure that Java 11 and Apache Maven are installed in your development environment.

- For Java, use Oracle Java SE Development Kit, Amazon Corretto, Red Hat OpenJDK, or AdoptOpenJDK.
- For Maven, go to [https://maven.apache.org/](https://maven.apache.org/).

**To create the deployment package (Java)**

1. Use the following command to create a new directory called `create-thumbnail` with a project configuration file (`pom.xml`) and a basic Java class.

   ```bash
   mvn -B archetype:generate \
   -DarchetypeGroupId=org.apache.maven.archetypes \
   -DgroupId=com.example.CreateThumbnail \
   -DartifactId=create-thumbnail
   ```

2. In the `create-thumbnail` directory that you created in the previous step, open the `pom.xml`. Replace its contents with the following code, and then save your changes.

   ```xml
   <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <!-- Your project details here -->
   </project>
   ```
Tutorial: Use an Amazon S3 trigger to create thumbnails
<build>
<plugins>
<plugin>
<artifactId>maven-surefire-plugin</artifactId>
/version>2.22.2</version>
</plugin>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-shade-plugin</artifactId>
<version>3.3.0</version>
<configuration>
<createDependencyReducedPom>false</createDependencyReducedPom>
</configuration>
<executions>
<execution>
<phase>package</phase>
<goals>
<goal>shade</goal>
</goals>
<configuration>
<filters>
<filter>
<artifact>*</artifact>
<excludes>
<exclude>META-INF/**</exclude>
<exclude>**/module-info.class</exclude>
</excludes>
</filter>
</filters>
</configuration>
</execution>
</executions>
</plugin>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>3.8.1</version>
<configuration>
<source>11</source>
<target>11</target>
</configuration>
</plugin>
</plugins>

Tutorial: Use an Amazon S3 trigger to create thumbnails
• The dependencyManagement section contains a dependency to the AWS SDK for Java.
• The dependencies section has a dependency for Amazon S3.
• The build section includes the Maven Shade plugin. This plugin creates a JAR file that contains the compiled function code and all of its dependencies. The Apache Maven Compiler Plugin is configured to use Java 11.

3. In the /create-thumbnail/src/main/java/com/example/CreateThumbnail directory, change the name of the App.java file to Handler.java.

4. Open Handler.java and replace its contents with the following code and save the file.

Example Handler.java

```java
package com.example.CreateThumbnail;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.image.BufferedImage;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import javax.imageio.ImageIO;
import software.amazon.awssdk.awscore.exception.AwsServiceException;
import software.amazon.awssdk.core.sync.RequestBody;
import software.amazon.awssdk.services.s3.model.GetObjectRequest;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;
import software.amazon.awssdk.services.s3.S3Client;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
```

Tutorial: Use an Amazon S3 trigger to create thumbnails
import com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNotificationRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

// Handler value: com.example.CreateThumbnail.Handler
public class Handler implements RequestHandler<S3Event, String> {
    Gson gson = new GsonBuilder().setPrettyPrinting().create();
    private static final float MAX_DIMENSION = 100;
    private final String REGEX = ".*\.\([^/\\.]*\)";
    private final String JPG_TYPE = "jpg";
    private final String JPG_MIME = "image/jpeg";
    private final String PNG_TYPE = "png";
    private final String PNG_MIME = "image/png";
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        final LambdaLogger logger = context.getLogger();
        try {
            logger.log("EVENT: " + gson.toJson(s3event));
            S3EventNotificationRecord record = s3event.getRecords().get(0);
            String srcBucket = record.getS3().getBucket().getName();
            // Object key may have spaces or unicode non-ASCII characters.
            String srcKey = record.getS3().getObject().getUrlDecodedKey();
            String dstBucket = srcBucket + "-resized";
            String dstKey = "resized-" + srcKey;
            // Infer the image type.
            Matcher matcher = Pattern.compile(REGEX).matcher(srcKey);
            if (!matcher.matches()) {
                logger.log("Unable to infer image type for key " + srcKey);
                return "";
            }
            String imageType = matcher.group(1);
            if (!(JPG_TYPE.equals(imageType)) && !(PNG_TYPE.equals(imageType))) {
                logger.log("Skipping non-image " + srcKey);
                return "";
            }
            // Download the image from S3 into a stream
            S3Client s3Client = S3Client.builder().build();
InputStream s3Object = getObject(s3Client, srcBucket, srcKey);

// Read the source image and resize it
BufferedImage srcImage = ImageIO.read(s3Object);
BufferedImage newImage = resizeImage(srcImage);

// Re-encode image to target format
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
ImageIO.write(newImage, imageType, outputStream);

// Upload new image to S3
putObject(s3Client, outputStream, dstBucket, dstKey, imageType, logger);

logger.log("Successfully resized " + srcBucket + "/" + srcKey + " and uploaded to " + dstBucket + "/" + dstKey);
return "Ok";
} catch (IOException e) {
    throw new RuntimeException(e);
}

private InputStream getObject(S3Client s3Client, String bucket, String key) {
    GetObjectRequest getObjectRequest = GetObjectRequest.builder()
        .bucket(bucket)
        .key(key)
        .build();
    return s3Client.getObject(getObjectRequest);
}

private void putObject(S3Client s3Client, ByteArrayOutputStream outputStream, String bucket, String key, String imageType, LambdaLogger logger) {
    Map<String, String> metadata = new HashMap<>();
    metadata.put("Content-Length", Integer.toString(outputStream.size()));
    if (JPG_TYPE.equals(imageType)) {
        metadata.put("Content-Type", JPG_MIME);
    } else if (PNG_TYPE.equals(imageType)) {
        metadata.put("Content-Type", PNG_MIME);
    }

    PutObjectRequest putObjectRequest = PutObjectRequest.builder()
        .bucket(bucket)
        .key(key)
        .metadata(metadata)
        .build();
}
// Uploading to S3 destination bucket
logger.log("Writing to: \" + bucket + "]\" + key);
try {
    s3Client.putObject(putObjectRequest,
        RequestBody.fromBytes(outputStream.toByteArray()));
} catch(AwsServiceException e) {
    logger.log(e.awsErrorDetails().errorMessage());
    System.exit(1);
}

/**
 * Resizes (shrinks) an image into a small, thumbnail-sized image.
 * The new image is scaled down proportionally based on the source
 * image. The scaling factor is determined based on the value of
 * MAX_DIMENSION. The resulting new image has max(height, width)
 * = MAX_DIMENSION.
 * @param srcImage BufferedImage to resize.
 * @return New BufferedImage that is scaled down to thumbnail size.
 */
private BufferedImage resizeImage(BufferedImage srcImage) {
    int srcHeight = srcImage.getHeight();
    int srcWidth = srcImage.getWidth();
    // Infer scaling factor to avoid stretching image unnaturally
    float scalingFactor = Math.min(
        MAX_DIMENSION / srcWidth, MAX_DIMENSION / srcHeight);
    int width = (int) (scalingFactor * srcWidth);
    int height = (int) (scalingFactor * srcHeight);

    BufferedImage resizedImage = new BufferedImage(width, height,
        BufferedImage.TYPE_INT_RGB);
    Graphics2D graphics = resizedImage.createGraphics();
    // Fill with white before applying semi-transparent (alpha) images
    graphics.setPaint(Color.white);
    graphics.fillRect(0, 0, width, height);
    // Simple bilinear resize
    graphics.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
        RenderingHints.VALUE_INTERPOLATION_BILINEAR);
    graphics.drawImage(srcImage, 0, 0, width, height, null);
graphics.dispose();
    return resizedImage;
  }
}

5. The `mvn -B archetype:generate` command from step 1 also generated a dummy test case in the `src/test` directory. For the purposes of this tutorial, skip over running tests by deleting this entire generated `/test` directory.

Note
As a best practice, always thoroughly test your code. For a working example of how you might unit test this application, see the `s3-java` example on GitHub.

6. Use the following command to build your project.

```
mvn package
```

This command generates a JAR file in the `target` directory.

Create the Lambda function

You can create your Lambda function using either the AWS CLI or the Lambda console. Follow the instructions for your chosen language to create the function.
To create the function (console)

To create your Lambda function using the console, you first create a basic function containing some ‘Hello world’ code. You then replace this code with your own function code by uploading the.zip or JAR file you created in the previous step.

1. Open the Functions page of the Lambda console.
2. Make sure you're working in the same AWS Region you created your Amazon S3 bucket in. You can change your region using the drop-down list at the top of the screen.
3. Choose Create function.
4. Choose Author from scratch.
5. Under Basic information, do the following:
   a. For Function name, enter CreateThumbnail.
   b. For Runtime choose either Node.js 18.x, Python 3.9, or Java 11 according to the language you chose for your function.
   c. For Architecture, choose x86_64.
6. In the Change default execution role tab, do the following:
   a. Expand the tab, then choose Use an existing role.
   b. Select the LambdaS3Role you created earlier.
7. Choose Create function.
To upload the function code (console)

1. In the Code source pane, choose Upload from.
2. For the Python and Node.js runtimes, choose .zip file. For the Java runtime, choose .zip or .jar file.
3. Choose Upload.
4. In the file selector, select your .zip or JAR file and choose Open.
5. Choose Save.

AWS CLI

To create the function (AWS CLI)

- Run the CLI command for the language you chose. For the role parameter, make sure to replace 123456789012 with your own AWS account ID. For the region parameter, replace us-west-2 with the region you created your Amazon S3 buckets in.

- For Node.js, run the following command from the directory containing your function.zip file.

```bash
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-west-2
```

- For Python, run the following command from the directory containing your lambda_function.zip file.

```bash
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://lambda_function.zip --handler lambda_function.lambda_handler \
--runtime python3.9 --timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-west-2
```

- For Java, run the following command from the target directory of your project.

```bash
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://create-thumbnail-1.0-SNAPSHOT.jar \
--handler com.example.CreateThumbnail.Handler \
```
For your Lambda function to run when you upload an image to your source bucket, you need to configure a trigger for your function. You can configure the Amazon S3 trigger using either the console or the AWS CLI.

⚠️ Important

This procedure configures the Amazon S3 bucket to invoke your function every time that an object is created in the bucket. Be sure to configure this only on the source bucket. If your Lambda function creates objects in the same bucket that invokes it, your function can be invoked continuously in a loop. This can result in unexpected charges being billed to your AWS account.

AWS Management Console

To configure the Amazon S3 trigger (console)

1. Open the Functions page of the Lambda console and choose your function (CreateThumbnail).
2. Choose Add trigger.
3. Select S3.
4. Under Bucket, select your source bucket.
5. Under Event types, select All object create events.

6. Under Recursive invocation, select the check box to acknowledge that using the same Amazon S3 bucket for input and output is not recommended. You can learn more about recursive invocation patterns in Lambda by reading Recursive patterns that cause run-away Lambda functions in Serverless Land.

7. Choose Add.

When you create a trigger using the Lambda console, Lambda automatically creates a resource based policy to give the service you select permission to invoke your function.

AWS CLI

To configure the Amazon S3 trigger (AWS CLI)

1. For your Amazon S3 source bucket to invoke your function when you add an image file, you first need to configure permissions for your function using a resource based policy. A resource-based policy statement gives other AWS services permission to invoke your function. To give Amazon S3 permission to invoke your function, run the following CLI command. Be sure to replace the source-account parameter with your own AWS account ID and to use your own source bucket name.

```bash
aws lambda add-permission --function-name CreateThumbnail \
--principal s3.amazonaws.com --statement-id s3invoke --action 
"lambda:InvokeFunction" \
--source-arn arn:aws:s3:::SOURCEBUCKET \
--source-account 123456789012
```

The policy you define with this command allows Amazon S3 to invoke your function only when an action takes place on your source bucket.

Note

Although Amazon S3 bucket names are globally unique, when using resource-based policies it is best practice to specify that the bucket must belong to your account. This is because if you delete a bucket, it is possible for another AWS account to create a bucket with the same Amazon Resource Name (ARN).
2. Save the following JSON in a file named `notification.json`. When applied to your source bucket, this JSON configures the bucket to send a notification to your Lambda function every time a new object is added. Replace the AWS account number and AWS Region in the Lambda function ARN with your own account number and region.

```json
{
    "LambdaFunctionConfigurations": [
        {
            "Id": "CreateThumbnailEventConfiguration",
            "Events": ["s3:ObjectCreated:Put"]
        }
    ]
}
```

3. Run the following CLI command to apply the notification settings in the JSON file you created to your source bucket. Replace `SOURCEBUCKET` with the name of your own source bucket.

```
aws s3api put-bucket-notification-configuration --bucket SOURCEBUCKET --notification-configuration file://notification.json
```

To learn more about the `put-bucket-notification-configuration` command and the `notification-configuration` option, see `put-bucket-notification-configuration` in the AWS CLI Command Reference.

**Test your Lambda function with a dummy event**

![Diagram of testing Lambda function with dummy event]
Before you test your whole setup by adding an image file to your Amazon S3 source bucket, you test that your Lambda function is working correctly by invoking it with a dummy event. An event in Lambda is a JSON-formatted document that contains data for your function to process. When your function is invoked by Amazon S3, the event sent to your function contains information such as the bucket name, bucket ARN, and object key.

AWS Management Console

**To test your Lambda function with a dummy event (console)**

1. Open the [Functions page](#) of the Lambda console and choose your function (CreateThumbnail).
2. Choose the Test tab.
3. To create your test event, in the Test event pane, do the following:
   a. Under Test event action, select Create new event.
   b. For Event name, enter myTestEvent.
   c. For Template, select S3 Put.
   d. Replace the values for the following parameters with your own values.
      - For awsRegion, replace us-east-1 with the AWS Region you created your Amazon S3 buckets in.
      - For name, replace example-bucket with the name of your own Amazon S3 source bucket.
      - For key, replace test%2Fkey with the filename of the test object you uploaded to your source bucket in the step [Upload a test image to your source bucket](#).

```json
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
    }
  ]
}
```
"requestParameters": {
  "sourceIPAddress": "127.0.0.1"
},
"responseElements": {
  "x-amz-request-id": "EXAMPLE123456789",
  "x-amz-id-2": "EXAMPLE123/5678abcdefgihijklambdaisawesome/mnopqrstuvwxyzABCDEFGH"
},
"s3": {
  "s3SchemaVersion": "1.0",
  "configurationId": "testConfigRule",
  "bucket": {
    "name": "example-bucket",
    "ownerIdentity": {
      "principalId": "EXAMPLE"
    },
    "arn": "arn:aws:s3:::example-bucket"
  },
  "object": {
    "key": "test%2Fkey",
    "size": 1024,
    "eTag": "0123456789abcdef0123456789abcdef",
    "sequencer": "0A1B2C3D4E5F678901"
  }
}
}

<table>
<thead>
<tr>
<th>e.</th>
<th>Choose <strong>Save</strong>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.</td>
<td>In the <strong>Test event</strong> pane, choose <strong>Test</strong>.</td>
</tr>
<tr>
<td>5.</td>
<td>To check the your function has created a resized verison of your image and stored it in your target Amazon S3 bucket, do the following:</td>
</tr>
<tr>
<td>a.</td>
<td>Open the <a href="https://aws.amazon.com">Buckets page</a> of the Amazon S3 console.</td>
</tr>
<tr>
<td>b.</td>
<td>Choose your target bucket and confirm that your resized file is listed in the <strong>Objects</strong> pane.</td>
</tr>
</tbody>
</table>
To test your Lambda function with a dummy event (AWS CLI)

1. Save the following JSON in a file named dummyS3Event.json. Replace the values for the following parameters with your own values:

   1. For `awsRegion`, replace `us-west-2` with the AWS Region you created your Amazon S3 buckets in.
   2. For `name`, replace `SOURCEBUCKET` with the name of your own Amazon S3 source bucket.
   3. For `key`, replace `HappyFace.jpg` with the filename of the test object you uploaded to your source bucket in the step [Upload a test image to your source bucket](#).

```json
{
   "Records": [
      {
         "eventVersion": "2.0",
         "eventSource": "aws:s3",
         "awsRegion": "us-west-2",
         "eventTime": "1970-01-01T00:00:00.000Z",
         "eventName": "ObjectCreated:Put",
         "userIdentity": {
            "principalId": "AIDAJDPLRKLG7UEXAMPLE"
         },
         "requestParameters": {
            "sourceIPAddress": "127.0.0.1"
         },
         "responseElements": {
            "x-amz-request-id": "C3D13FE58DE4C810",
            "x-amz-id-2": "FMyUVURIY8/IgAtTv8xRjskZQpcIZ9KG4V5Wp6S7S/JRWeUWerMUE5JgHvANOjpD"
         },
         "s3": {
            "s3SchemaVersion": "1.0",
            "configurationId": "testConfigRule",
            "bucket": {
               "name": "SOURCEBUCKET",
               "ownerIdentity": {
                  "principalId": "A3NL1K0ZKExampfl"
               },
               "arn": "arn:aws:s3:::SOURCEBUCKET"
            }
         }
      }
   ]
}
```
2. From the directory you saved your `dummyS3Event.json` file in, invoke the function by running the following CLI command. This command invokes your Lambda function synchronously by specifying RequestResponse as the value of the invocation-type parameter. To learn more about synchronous and asynchronous invocation, see Invoking Lambda functions.

```bash
aws lambda invoke --function-name CreateThumbnail \
--invocation-type RequestResponse --cli-binary-format raw-in-base64-out \
--payload file://dummyS3Event.json outputfile.txt
```

The cli-binary-format option is required if you are using version 2 of the AWS CLI. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options.

3. Verify that your function has created a thumbnail version of your image and saved it to your target Amazon S3 bucket. Run the following CLI command, replacing `SOURCEBUCKET-resized` with the name of your own destination bucket.

```bash
aws s3api list-objects-v2 --bucket SOURCEBUCKET-resized
```

You should see output similar to the following. The Key parameter shows the filename of your resized image file.

```json
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-06T21:40:07+00:00",
            "ETag": "\"d8ca652ffe83ba6b721fffc20d9d7174a\"",
```
Test your function using the Amazon S3 trigger

AWS Management Console

**To test your Lambda function using the Amazon S3 trigger (console)**

1. To upload an image to your Amazon S3 bucket, do the following:
   a. Open the [Buckets](#) page of the Amazon S3 console and choose your source bucket.
   b. Choose [Upload](#).
   c. Choose [Add files](#) and use the file selector to choose the image file you want to upload. Your image object can be any .jpg or .png file.
   d. Choose [Open](#), then choose [Upload](#).
2. Verify that Lambda has saved a resized version of your image file in your target bucket by doing the following:
a. Navigate back to the Buckets page of the Amazon S3 console and choose your destination bucket.

b. In the Objects pane, you should now see two resized image files, one from each test of your Lambda function. To download your resized image, select the file, then choose Download.

AWS CLI

To test your Lambda function using the Amazon S3 trigger (AWS CLI)

1. From the directory containing the image you want to upload, run the following CLI command. Replace the --bucket parameter with the name of your source bucket. For the --key and --body parameters, use the filename of your test image. Your test image can be any .jpg or .png file.

```bash
aws s3api put-object --bucket SOURCEBUCKET --key SmileyFace.jpg --body ./SmileyFace.jpg
```

2. Verify that your function has created a thumbnail version of your image and saved it to your target Amazon S3 bucket. Run the following CLI command, replacing SOURCEBUCKET-resized with the name of your own destination bucket.

```bash
aws s3api list-objects-v2 --bucket SOURCEBUCKET-resized
```

If your function runs successfully, you’ll see output similar to the following. Your target bucket should now contain two resized files.

```json
{
   "Contents": [
      {
         "Key": "resized-HappyFace.jpg",
         "LastModified": "2023-06-07T00:15:50+00:00",
         "ETag": "7781a43e765a8301713f533d70968a1e",
         "Size": 2763,
         "StorageClass": "STANDARD"
      },
      {
         "Key": "resized-SmileyFace.jpg",
         "LastModified": "2023-06-07T00:13:18+00:00",
```
Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose Actions, Delete.
4. Type delete in the text input field and choose Delete.

To delete the policy that you created

1. Open the Policies page of the IAM console.
2. Select the policy that you created (AWSLambdaS3Policy).
3. Choose Policy actions, Delete.
4. Choose Delete.

To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose Delete.
4. Enter the name of the role in the text input field and choose Delete.
To delete the S3 bucket

1. Open the Amazon S3 console.
2. Select the bucket you created.
3. Choose **Delete**.
4. Enter the name of the bucket in the text input field.
5. Choose **Delete bucket**.
Using AWS Lambda with Amazon S3 batch operations

You can use Amazon S3 batch operations to invoke a Lambda function on a large set of Amazon S3 objects. Amazon S3 tracks the progress of batch operations, sends notifications, and stores a completion report that shows the status of each action.

To run a batch operation, you create an Amazon S3 batch operations job. When you create the job, you provide a manifest (the list of objects) and configure the action to perform on those objects.

When the batch job starts, Amazon S3 invokes the Lambda function synchronously for each object in the manifest. The event parameter includes the names of the bucket and the object.

The following example shows the event that Amazon S3 sends to the Lambda function for an object that is named customerImage1.jpg in the examplebucket bucket.

**Example Amazon S3 batch request event**

```
{
  "invocationSchemaVersion": "1.0",
  "invocationId": "YXNkbGZqYWRmaiBhc2RmdW9hZHNgZGpmaGFzDgtaGZza2RmaAo",
  "job": {
    "id": "f3cc4f60-61f6-4a2b-8a21-d07600c373ce"
  },
  "tasks": [
    {
      "taskId": "dGFza2lkZ291c2h1cmUK",
      "s3Key": "customerImage1.jpg",
      "s3VersionId": "1",
      "s3BucketArn": "arn:aws:s3:::examplebucket"
    }
  ]
}
```

Your Lambda function must return a JSON object with the fields as shown in the following example. You can copy the invocationId and taskId from the event parameter. You can return a string in the resultString. Amazon S3 saves the resultString values in the completion report.

**Example Amazon S3 batch request response**

```json
S3 Batch
```
Invoking Lambda functions from Amazon S3 batch operations

You can invoke the Lambda function with an unqualified or qualified function ARN. If you want to use the same function version for the entire batch job, configure a specific function version in the FunctionARN parameter when you create your job. If you configure an alias or the $LATEST qualifier, the batch job immediately starts calling the new version of the function if the alias or $LATEST is updated during the job execution.

Note that you can't reuse an existing Amazon S3 event-based function for batch operations. This is because the Amazon S3 batch operation passes a different event parameter to the Lambda function and expects a return message with a specific JSON structure.

In the resource-based policy that you create for the Amazon S3 batch job, ensure that you set permission for the job to invoke your Lambda function.

In the execution role for the function, set a trust policy for Amazon S3 to assume the role when it runs your function.

If your function uses the AWS SDK to manage Amazon S3 resources, you need to add Amazon S3 permissions in the execution role.

When the job runs, Amazon S3 starts multiple function instances to process the Amazon S3 objects in parallel, up to the concurrency limit of the function. Amazon S3 limits the initial ramp-up of instances to avoid excess cost for smaller jobs.

If the Lambda function returns a TemporaryFailure response code, Amazon S3 retries the operation.
For more information about Amazon S3 batch operations, see [Performing batch operations](#) in the Amazon S3 Developer Guide.

For an example of how to use a Lambda function in Amazon S3 batch operations, see [Invoking a Lambda function from Amazon S3 batch operations](#) in the Amazon S3 Developer Guide.
Transforming S3 Objects with S3 Object Lambda

With S3 Object Lambda you can add your own code to Amazon S3 GET, HEAD, and LIST requests to modify and process data before it is returned to an application. You can use custom code to modify the data returned by standard S3 GET, HEAD, or LIST requests to filter rows, dynamically resize images, redact confidential data, and more. Powered by AWS Lambda functions, your code runs on infrastructure that is fully managed by AWS, eliminating the need to create and store derivative copies of your data or to run proxies, all with no changes required to applications.

For more information, see Transforming objects with S3 Object Lambda.

Tutorials

- Transforming data for your application with Amazon S3 Object Lambda
- Detecting and redacting PII data with Amazon S3 Object Lambda and Amazon Comprehend
- Using Amazon S3 Object Lambda to Dynamically Watermark Images as They Are Retrieved
Using AWS Lambda with Secrets Manager

Your AWS Lambda function can interact with AWS Secrets Manager using the Secrets Manager API or any of the AWS Software Development Kits (SDKs). You can also use the AWS Parameters and Secrets Lambda Extension to retrieve and cache AWS Secrets Manager secrets in Lambda functions without using an SDK. See Use AWS Secrets Manager secrets in AWS Lambda functions for more information.
Using AWS Lambda with Amazon SES

When you use Amazon SES to receive messages, you can configure Amazon SES to call your Lambda function when messages arrive. The service can then invoke your Lambda function by passing in the incoming email event, which in reality is an Amazon SES message in an Amazon SNS event, as a parameter.

Example Amazon SES message event

```
{
   "Records": [
      {
         "eventVersion": "1.0",
         "ses": {
            "mail": {
               "commonHeaders": {
                  "from": ["Jane Doe <janedoe@example.com>"],
                  "to": ["johndoe@example.com"],
                  "returnPath": "janedoe@example.com",
                  "messageId": "<0123456789example.com>",
                  "date": "Wed, 7 Oct 2015 12:34:56 -0700",
                  "subject": "Test Subject"
               },
               "source": "janedoe@example.com",
               "timestamp": "1970-01-01T00:00:00.000Z",
               "destination": ["johndoe@example.com"],
               "headers": [
                  {
                     "name": "Return-Path",
                     "value": "<janedoe@example.com>"
                  },
                  {
                     "name": "Received",
                     "value": "from mailer.example.com (mailer.example.com [203.0.113.1]) by inbound-smtp.us-west-2.amazonaws.com with SMTP id o3vri10e2ic for johndoe@example.com; Wed, 07 Oct 2015 12:34:56 +0000 (UTC)"
                  }
               ]
            }
         }
      }
   ]
}
```
{  
    "name": "DKIM-Signature",
    "value": "v=1; a=rsa-sha256; c=relaxed/relaxed; d=example.com;  
s=example; h=mime-version:from:date:message-id:subject:to:content-type;  
bh=jX3F0bCAI7sIbkHyy3mLY028ieDQz2R0P8HwQkk1Fj4=; b=sQwJ+LMe9RjkesGu  
+vqU56asvMhrLRRYrWcbV"
},
{
    "name": "MIME-Version",
    "value": "1.0"
},
{
    "name": "From",
    "value": "Jane Doe <janedoe@example.com>"
},
{
    "name": "Date",
    "value": "Wed, 7 Oct 2015 12:34:56 -0700"
},
{
    "name": "Message-ID",
    "value": "<0123456789@example.com>"
},
{
    "name": "Subject",
    "value": "Test Subject"
},
{
    "name": "To",
    "value": "johndoe@example.com"
},
{
    "name": "Content-Type",
    "value": "text/plain; charset=UTF-8"
}
],
"headersTruncated": false,
"messageId": "o3vrnil0e2ic28tr"
",
"receipt": {
    "recipients": [
        "johndoe@example.com"
    ],
    "timestamp": "1970-01-01T00:00:00.000Z"
"spamVerdict": {  
   "status": "PASS"
 },
"dkimVerdict": {  
   "status": "PASS"
 },
"processingTimeMillis": 574,
"action": {  
   "type": "Lambda",
   "invocationType": "Event",
   "functionArn": "arn:aws:lambda:us-west-2:111122223333:function:Example"
 },
"spfVerdict": {  
   "status": "PASS"
 },
"virusVerdict": {  
   "status": "PASS"
 }
],
"eventSource": "aws:ses"
}]

For more information, see Lambda action in the Amazon SES Developer Guide.
Using AWS Lambda with Amazon SNS

You can use a Lambda function to process Amazon Simple Notification Service (Amazon SNS) notifications. Amazon SNS supports Lambda functions as a target for messages sent to a topic. You can subscribe your function to topics in the same account or in other AWS accounts.

Amazon SNS invokes your function asynchronously with an event that contains a message and metadata.

Example Amazon SNS message event

```json
{
    "Records": [
        {
            "EventVersion": "1.0",
            "EventSource": "aws:sns",
            "Sns": {
                "SignatureVersion": "1",
                "Timestamp": "2019-01-02T12:45:07.000Z",
                "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEki6RibDsvpi+tE/1+82j...65r==",
                "SigningCertUrl": "https://sns.us-east-1.amazonaws.com/SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
                "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
                "Message": "Hello from SNS!",
                "MessageAttributes": {
                    "Test": {
                        "Type": "String",
                        "Value": "TestString"
                    },
                    "TestBinary": {
                        "Type": "Binary",
                        "Value": "TestBinary"
                    }
                },
                "Type": "Notification",
                "Subject": "TestInvoke"
            }
        }
    ]
}
```
For asynchronous invocation, Lambda queues the message and handles retries. If Amazon SNS can't reach Lambda or the message is rejected, Amazon SNS retries at increasing intervals over several hours. For details, see Reliability in the Amazon SNS FAQs.

To perform cross-account Amazon SNS deliveries to Lambda, you must authorize Amazon SNS to invoke your Lambda function. In turn, Amazon SNS must allow the AWS account with the Lambda function to subscribe to the Amazon SNS topic. For example, if the Amazon SNS topic is in account A and the Lambda function is in account B, both accounts must grant permissions to the other to access their respective resources. Since not all the options for setting up cross-account permissions are available from the AWS Management Console, you must use the AWS Command Line Interface (AWS CLI) for setup.

For more information, see Fanout to Lambda functions in the Amazon Simple Notification Service Developer Guide.

**Input types for Amazon SNS events**

For input type examples for Amazon SNS events in Java, .NET, and Go, see the following on the AWS GitHub repository:

- SNSEvent.java
- SNSEvent.cs
- sns.go

**Topics**

- Tutorial: Using AWS Lambda with Amazon Simple Notification Service
- Sample function code

**Tutorial: Using AWS Lambda with Amazon Simple Notification Service**

In this tutorial, you use a Lambda function in one AWS account to subscribe to an Amazon Simple Notification Service (Amazon SNS) topic in a separate AWS account. When you publish messages to your Amazon SNS topic, your Lambda function reads the contents of the message and outputs it
To Amazon CloudWatch Logs. To complete this tutorial, you use the AWS Command Line Interface (AWS CLI).

To complete this tutorial, you perform the following steps:

- In **account A**, create an Amazon SNS topic.
- In **account B**, create a Lambda function that will read messages from the topic.
- In **account B**, create a subscription to the topic.
- Publish messages to the Amazon SNS topic in **account A** and confirm that the Lambda function in **account B** outputs them to CloudWatch Logs.

By completing these steps, you will learn how to configure an Amazon SNS topic to invoke a Lambda function. You will also learn how to create an AWS Identity and Access Management (IAM) policy that gives permission for a resource in another AWS account to invoke Lambda.

In the tutorial, you use two separate AWS accounts. The AWS CLI commands illustrate this by using two named profiles called accountA and accountB, each configured for use with a different AWS account. To learn how to configure the AWS CLI to use different profiles, see [Configuration and credential file settings](#) in the *AWS Command Line Interface User Guide for Version 2*. Be sure to configure the same default AWS Region for both profiles.

If the AWS CLI profiles you create for the two AWS accounts use different names, or if you use the default profile and one named profile, modify the AWS CLI commands in the following steps as needed.
Prerequisites

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

When you sign up for an AWS account, an AWS account root user is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to an administrative user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing My Account.

Create an administrative user

After you sign up for an AWS account, create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing Root user and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the AWS Sign-In User Guide.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the IAM User Guide.
Create an administrative user

- For your daily administrative tasks, grant administrative access to an administrative user in AWS IAM Identity Center.

  For instructions, see Getting started in the AWS IAM Identity Center User Guide.

Sign in as the administrative user

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the AWS Sign-In User Guide.

Install the AWS Command Line Interface

If you have not yet installed the AWS Command Line Interface, follow the steps at Installing or updating the latest version of the AWS CLI to install it.

The tutorial requires a command line terminal or shell to run commands. In Linux and macOS, use your preferred shell and package manager.

ℹ️ Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux.
Create an Amazon SNS topic (account A)

To create the topic

- In account A, create an Amazon SNS standard topic using the following AWS CLI command.

  `aws sns create-topic --name sns-topic-for-lambda --profile accountA`

  You should see output similar to the following.

  ```
  {
  }
  ```

  Make a note of the Amazon Resource Name (ARN) of your topic. You'll need it later in the tutorial when you add permissions to your Lambda function to subscribe to the topic.

Create a function execution role (account B)

An execution role is an IAM role that grants a Lambda function permission to access AWS services and resources. Before you create your function in account B, you create a role that gives the
function basic permissions to write logs to CloudWatch Logs. We’ll add the permissions to read from your Amazon SNS topic in a later step.

To create an execution role

1. In account B open the roles page in the IAM console.
2. Choose Create role.
3. For Trusted entity type, choose AWS service.
4. For Use case, choose Lambda.
5. Choose Next.
6. Add a basic permissions policy to the role by doing the following:
   a. In the Permissions policies search box, enter AWSLambdaBasicExecutionRole.
   b. Choose Next.
7. Finalize the role creation by doing the following:
   a. Under Role details, enter lambda-sns-role for Role name.
   b. Choose Create role.

Create a Lambda function (account B)

Create a Lambda function that processes your Amazon SNS messages. The function code logs the message contents of each record to Amazon CloudWatch Logs.

This tutorial uses the Node.js 18.x runtime, but we’ve also provided example code in other runtime languages. You can select the tab in the following box to see code for the runtime you’re interested in. The JavaScript code you’ll use in this step is in the first example shown in the JavaScript tab.
Consuming an SNS event with Lambda using .NET.

```csharp
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
[assembly:
    LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer))]
namespace SnsIntegration;
public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record, ILambdaContext context)
    {
        try
        {
            // TODO: Do interesting work based on the new message
        } finally
        {
            context.Logger.LogInformation("done");
        }
    }
}
```
await Task.CompletedTask;
}
catch (Exception e)
{
  // You can use Dead Letter Queue to handle failures. By configuring a Lambda DLQ.
  context.Logger.LogError($"An error occurred");
  throw;
}
}

---

**Go**

**SDK for Go V2**

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](https://github.com/aws/aws-lambda-go) repository.

---

Consuming an SNS event with Lambda using Go.

```go
package main

import (
  "context"
  "fmt"
  "github.com/aws/aws-lambda-go/events"
  "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
  for _, record := range snsEvent.Records {
    processMessage(record)
  }
  fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
```
message := record.SNS.Message
fmt.Printf("Processed message: %s\n", message)
// TODO: Process your record here
}

cfunc main() {
    lambda.Start(handler)
}

JavaScript

SDK for JavaScript (v2)

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using JavaScript.

exports.handler = async (event, context) => {
    for (const record of event.Records) {
        await processMessageAsync(record);
    }
    console.info("done");
};

async function processMessageAsync(record) {
    try {
        const message = JSON.stringify(record.Sns.Message);
        console.log('Processed message ${message}');
        await Promise.resolve(1); //Placeholder for actual async work
    } catch (err) {
        console.error("An error occurred");
        throw err;
    }
}
Consuming an SNS event with Lambda using TypeScript.

```typescript
import { SNSEvent, Context, SNSHandler, SNEVENTRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (event: SNSEvent, context: Context): Promise<void> => {
    for (const record of event.Records) {
        await processMessageAsync(record);
    }
    console.info("done");
};

async function processMessageAsync(record: SNEVENTRecord): Promise<any> {
    try {
        const message: string = JSON.stringify(record.Sns.Message);
        console.log("Processed message \${message}\n");
        await Promise.resolve(1); // Placeholder for actual async work
    } catch (err) {
        console.error("An error occurred");
        throw err;
    }
}
```

Python

**SDK for Python (Boto3)**

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming an SNS event with Lambda using Python.

```python
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
```
print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}"")
        # TODO; Process your record here
    except Exception as e:
        print("An error occurred")
        raise e

To create the function

1. Create a directory for the project, and then switch to that directory.

mkdir sns-tutorial

cd sns-tutorial

2. Copy the sample JavaScript code into a new file named index.js.

3. Create a deployment package using the following zip command.

zip function.zip index.js

4. Run the following AWS CLI command to create your Lambda function in account B.

aws lambda create-function --function-name Function-With-SNS \\
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \\
--role arn:aws:iam::<AccountB_ID>:role/lambda-sns-role \\
--timeout 60 --profile accountB

You should see output similar to the following.

{  "FunctionName": "Function-With-SNS",
   "Runtime": "nodejs18.x",
   "Role": "arn:aws:iam::123456789012:role/lambda_basic_role",
   "Handler": "index.handler"}
5. Record the Amazon Resource Name (ARN) of your function. You’ll need it later in the tutorial when you add permissions to allow Amazon SNS to invoke your function.

**Add permissions to function (account B)**

For Amazon SNS to invoke your function, you need to grant it permission in a statement on a resource-based policy. You add this statement using the AWS CLI add-permission command.

**To grant Amazon SNS permission to invoke your function**

- In **account B**, run the following AWS CLI command using the ARN for your Amazon SNS topic you recorded earlier.

```
aws lambda add-permission --function-name Function-With-SNS  
--source-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda  
--statement-id function-with-sns --action "lambda:InvokeFunction"  
--principal sns.amazonaws.com --profile accountB
```

You should see output similar to the following.

```json
{
   "Statement": "{
       "Condition": {"ArnLike": {"AWS:SourceArn": "arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda"}},
       "Action": ["lambda:InvokeFunction"]
   }
}
```
Grant cross-account permission for Amazon SNS subscription (account A)

For your Lambda function in account B to subscribe to the Amazon SNS topic you created in account A, you need to grant permission for account B to subscribe to your topic. You grant this permission using the AWS CLI add-permission command.

To grant permission for account B to subscribe to the topic

- In account A, run the following AWS CLI command. Use the ARN for the Amazon SNS topic you recorded earlier.

```bash
aws sns add-permission --label lambda-access --aws-account-id <AccountB_ID> \
--topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
--action-name Subscribe ListSubscriptionsByTopic --profile accountA
```
Create a subscription (account B)

In **account B**, you now subscribe your Lambda function to the Amazon SNS topic you created at the beginning of the tutorial in **account A**. When a message is sent to this topic (**sns-topic-for-lambda**), Amazon SNS invokes your Lambda function **Function-With-SNS** in **account B**.

**To create a subscription**

- In **account B**, run the following AWS CLI command. Use your default region you created your topic in and the ARNs for your topic and Lambda function.

```bash
aws sns subscribe --protocol lambda \
  --region us-east-1 \
  --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
  --profile accountB
```

You should see output similar to the following.

```json
{
   "SubscriptionArn": "arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda:5d906xxxx-7c8x-45dx-a9dx-0484e31c98xx"
}
```
Publish messages to topic (account A and account B)

Now that your Lambda function in **account B** is subscribed to your Amazon SNS topic in **account A**, it's time to test your setup by publishing messages to your topic. To confirm that Amazon SNS has invoked your Lambda function, you use CloudWatch Logs to view your function's output.

**To publish a message to your topic and view your function's output**

1. Enter Hello World into a text file and save it as `message.txt`.
2. From the same directory you saved your text file in, run the following AWS CLI command in **account A**. Use the ARN for your own topic.

   ```bash
   aws sns publish --message file://message.txt --subject Test \
   --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
   --profile accountA
   ```

   This will return a message ID with a unique identifier, indicating that Amazon SNS has accepted the message. Amazon SNS then attempts to deliver the message to the topic's subscribers. To confirm that Amazon SNS has invoked your Lambda function, use CloudWatch Logs to view your function's output:

3. **In account B**, open the [Log groups](https://aws.amazon.com/cloudwatch/logs) page of the Amazon CloudWatch console.
4. Choose the log group for your function (/aws/lambda/Function-With-SNS).
5. Choose the most recent log stream.
6. If your function was correctly invoked, you'll see output similar to the following showing the contents of the message you published to your topic.

   ```log
   2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO Processed message Hello World
   2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO done
   ```
Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

In **Account A**, clean up your Amazon SNS topic.

**To delete the Amazon SNS topic**

1. Open the [Topics page](#) of the Amazon SNS console.
2. Select the topic you created.
3. Choose **Delete**.
4. Enter **delete me** in the text input field.
5. Choose **Delete**.

In **Account B**, clean up your execution role, Lambda function, and Amazon SNS subscription.

**To delete the execution role**

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

**To delete the Lambda function**

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

**To delete the Amazon SNS subscription**

1. Open the [Subscriptions page](#) of the Amazon SNS console.
2. Select the subscription you created.

3. Choose **Delete, Delete.**

**Sample function code**

Sample code is available for the following languages.

Topics

- **Node.js 12.x**
- **Java 11**
- **Go**
- **Python 3**

**Node.js 12.x**

The following example processes a message from Amazon SNS and sends the content of that message to the logs.

**Example index.js**

```javascript
console.log('Loading function');

exports.handler = function(event, context, callback) {
  // console.log('Received event:', JSON.stringify(event, null, 4));

  var message = event.Records[0].Sns.Message;
  console.log('Message received from SNS:', message);
  callback(null, "Success");
};
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Node.js Lambda functions with .zip file archives](#).

**Java 11**

The following example processes a message from Amazon SNS and sends the content of that message to the logs.
Example LogEvent.java

```java
package example;

import java.text.SimpleDateFormat;
import java.util.Calendar;

import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;

public class LogEvent implements RequestHandler<SNSEvent, Object> {
    public Object handleRequest(SNSEvent request, Context context) {
        String timeStamp = new SimpleDateFormat("yyyy-MM-dd_HH:mm:ss").format(Calendar.getInstance().getTime());
        context.getLogger().log("Invocation started: " + timeStamp);
        context.getLogger().log(request.getRecords().get(0).getSNS().getMessage());

        timeStamp = new SimpleDateFormat("yyyy-MM-dd_HH:mm:ss").format(Calendar.getInstance().getTime());
        context.getLogger().log("Invocation completed: " + timeStamp);
        return null;
    }
}
```

Dependencies

- aws-lambda-java-core
- aws-lambda-java-events

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [Deploy Java Lambda functions with .zip or JAR file archives](#).

Go

The following example processes a message from Amazon SNS and sends the content of that message to the logs.

Example lambda_handler.go

```go
package main
```
import (
    "context"
    "fmt"
    "github.com/aws/aws-lambda-go/lambda"
    "github.com/aws/aws-lambda-go/events"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
    for _, record := range snsEvent.Records {
        snsRecord := record.SNS
        fmt.Printf("[%s %s] Message = %s \n", record.EventSource, snsRecord.Timestamp,
                 snsRecord.Message)
    }
}

func main() {
    lambda.Start(handler)
}

Build the executable with `go build` and create a deployment package. For instructions, see [Deploy Go Lambda functions with .zip file archives](#).

**Python 3**

The following example processes a message from Amazon SNS and sends the content of that message to the logs.

**Example lambda_handler.py**

```python
from __future__ import print_function
import json
print('Loading function')

def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))
    message = event['Records'][0]['Sns']['Message']
    print("From SNS: " + message)
    return message

Zip up the sample code to create a deployment package. For instructions, see [Working with .zip file archives for Python Lambda functions](#).
Using Lambda with Amazon SQS

Note
If you want to send data to a target other than a Lambda function or enrich the data before sending it, see Amazon EventBridge Pipes.

You can use a Lambda function to process messages in an Amazon Simple Queue Service (Amazon SQS) queue. Lambda event source mappings support standard queues and first-in, first-out (FIFO) queues. With Amazon SQS, you can offload tasks from one component of your application by sending them to a queue and processing them asynchronously.

Lambda polls the queue and invokes your Lambda function synchronously with an event that contains queue messages. Lambda reads messages in batches and invokes your function once for each batch. When your function successfully processes a batch, Lambda deletes its messages from the queue.

When Lambda reads a batch, the messages stay in the queue but are hidden for the length of the queue's visibility timeout. If your function successfully processes the batch, Lambda deletes the messages from the queue. By default, if your function encounters an error while processing a batch, all messages in that batch become visible in the queue again. For this reason, your function code must be able to process the same message multiple times without unintended side effects.

To prevent Lambda from processing a message multiple times, you can either configure your event source mapping to include batch item failures in your function response, or you can use the Amazon SQS API action DeleteMessage to remove messages from the queue as your Lambda function successfully processes them. For more information on using the Amazon SQS API, see the Amazon Simple Queue Service API Reference.

Example standard queue message event

Example Amazon SQS message event (standard queue)

```json
{
  "Records": [ 
    {
      "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
      "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlSa3SLy0a...",
```
By default, Lambda polls up to 10 messages in your queue at once and sends that batch to your function. To avoid invoking the function with a small number of records, you can tell the event source to buffer records for up to 5 minutes by configuring a batch window. Before invoking the function, Lambda continues to poll messages from the SQS standard queue until the batch window expires, the invocation payload size quota is reached, or the configured maximum batch size is reached.

If you're using a batch window and your SQS queue contains very low traffic, Lambda might wait for up to 20 seconds before invoking your function. This is true even if you set a batch window lower than 20 seconds.
Note

In Java, you might experience null pointer errors when deserializing JSON. This could be due to how case of "Records" and "eventSourceARN" is converted by the JSON object mapper.

Example FIFO queue message event

For FIFO queues, records contain additional attributes that are related to deduplication and sequencing.

Example Amazon SQS message event (FIFO queue)

```json
{
   "Records": [
      {
         "messageId": "11d6ee51-4cc7-4302-9e22-7cd8afdaadf5",
         "receiptHandle": "AQEBBX8nesZExmkmzeyIE8iQAMig7qw...",
         "body": "Test message.",
         "attributes": {
            "ApproximateReceiveCount": "1",
            "SentTimestamp": "1573251510774",
            "SequenceNumber": "18849496460467696128",
            "MessageGroupId": "1",
            "SenderId": "AIDAIO23YVJENQ2JOL4VO",
            "MessageDeduplicationId": "1",
            "ApproximateFirstReceiveTimestamp": "1573251510774"
         },
         "messageAttributes": {},
         "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
         "eventSource": "aws:sqs",
         "awsRegion": "us-east-2"
      }
   ]
}
```
Configuring a queue to use with Lambda

Create an SQS queue to serve as an event source for your Lambda function. Then configure the queue to allow time for your Lambda function to process each batch of events—and for Lambda to retry in response to throttling errors as it scales up.

To allow your function time to process each batch of records, set the source queue's visibility timeout to at least six times the timeout that you configure on your function. The extra time allows for Lambda to retry if your function is throttled while processing a previous batch.

If your function fails to process a message multiple times, Amazon SQS can send it to a dead-letter queue. If your function returns an error, all items in the batch return to the queue. After the visibility timeout occurs, Lambda receives the message again. To send messages to a second queue after a number of receives, configure a dead-letter queue on your source queue.

Note

Make sure that you configure the dead-letter queue on the source queue, not on the Lambda function. The dead-letter queue that you configure on a function is used for the function's asynchronous invocation queue, not for event source queues.

If your function returns an error, or can't be invoked because it's at maximum concurrency, processing might succeed with additional attempts. To give messages a better chance to be processed before sending them to the dead-letter queue, set the maxReceiveCount on the source queue's redrive policy to at least 5.

Execution role permissions

Lambda needs the following permissions to manage messages in your Amazon SQS queue. Add them to your function's execution role.

- sqs:ReceiveMessage
- sqs:DeleteMessage
- sqs:GetQueueAttributes

Optionally, if you're using an encrypted queue, you also need to add the following permission to your execution role:
Configuring a queue as an event source

Create an event source mapping to tell Lambda to send items from your queue to a Lambda function. You can create multiple event source mappings to process items from multiple queues with a single function. When Lambda invokes the target function, the event can contain multiple items, up to a configurable maximum batch size.

To configure your function to read from Amazon SQS in the Lambda console, create an SQS trigger.

To create a trigger

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
4. Choose the SQS trigger type.
5. Configure the required options, and then choose Add.

Lambda supports the following options for Amazon SQS event sources:

**SQS queue**

The Amazon SQS queue to read records from.

**Enable trigger**

The status of the event source mapping. Enable trigger is selected by default.

**Batch size**

The maximum number of records to send to the function in each batch. For a standard queue, this can be up to 10,000 records. For a FIFO queue, the maximum is 10. For a batch size over 10, you must also set the batch window (MaximumBatchingWindowInSeconds) to at least 1 second.

Configure your function timeout to allow enough time to process an entire batch of items. If items take a long time to process, choose a smaller batch size. A large batch size can improve efficiency for workloads that are very fast or have a lot of overhead. If you configure reserved...
concurrency on your function, set a minimum of five concurrent executions to reduce the chance of throttling errors when Lambda invokes your function.

Lambda passes all of the records in the batch to the function in a single call, as long as the total size of the events doesn't exceed the invocation payload size quota for synchronous invocation (6 MB). Both Lambda and Amazon SQS generate metadata for each record. This additional metadata is counted towards the total payload size and can cause the total number of records sent in a batch to be lower than your configured batch size. The metadata fields that Amazon SQS sends can be variable in length. For more information about the Amazon SQS metadata fields, see the ReceiveMessage API operation documentation in the Amazon Simple Queue Service API Reference.

**Batch window**

The maximum amount of time to gather records before invoking the function, in seconds. This applies only to standard queues.

If you're using a batch window greater than 0 seconds, you must account for the increased processing time in your queue's visibility timeout. We recommend setting your queue's visibility timeout to six times your function timeout, plus the value of MaximumBatchingWindowInSeconds. This allows time for your Lambda function to process each batch of events and to retry in the event of a throttling error.

When messages become available, Lambda starts processing messages in batches. Lambda starts processing five batches at a time with five concurrent invocations of your function. If messages are still available, Lambda adds up to 300 more instances of your function a minute, up to a maximum of 1,000 function instances. To learn more about function scaling and concurrency, see Lambda function scaling.

To process more messages, you can optimize your Lambda function for higher throughput. See Understanding how AWS Lambda scales with Amazon SQS standard queues.

**Maximum concurrency**

The maximum number of concurrent functions that the event source can invoke. For more information, see Configuring maximum concurrency for Amazon SQS event sources.

**Filter criteria**

Add filter criteria to control which events Lambda sends to your function for processing. For more information, see Lambda event filtering.
Scaling and processing

For standard queues, Lambda uses long polling to poll a queue until it becomes active. When messages are available, Lambda starts processing five batches at a time with five concurrent invocations of your function. If messages are still available, Lambda increases the number of processes that are reading batches by up to 300 more instances per minute. The maximum number of batches that an event source mapping can process simultaneously is 1,000.

For FIFO queues, Lambda sends messages to your function in the order that it receives them. When you send a message to a FIFO queue, you specify a message group ID. Amazon SQS ensures that messages in the same group are delivered to Lambda in order. Lambda sorts the messages into groups and sends only one batch at a time for a group. If your function returns an error, the function attempts all retries on the affected messages before Lambda receives additional messages from the same group.

Your function can scale in concurrency to the number of active message groups. For more information, see SQS FIFO as an event source on the AWS Compute Blog.

Configuring maximum concurrency for Amazon SQS event sources

The maximum concurrency setting limits the number of concurrent instances of the function that an Amazon SQS event source can invoke. Maximum concurrency is an event source-level setting. If you have multiple Amazon SQS event sources mapped to one function, each event source can have a separate maximum concurrency setting. You can use maximum concurrency to prevent one queue from using all of the function's reserved concurrency or the rest of the account's concurrency quota. There is no charge for configuring maximum concurrency on an Amazon SQS event source.

Importantly, maximum concurrency and reserved concurrency are two independent settings. Don't set maximum concurrency higher than the function's reserved concurrency. If you configure maximum concurrency, make sure that your function's reserved concurrency is greater than or equal to the total maximum concurrency for all Amazon SQS event sources on the function. Otherwise, Lambda may throttle your messages.

If maximum concurrency is not set, Lambda can scale your Amazon SQS event source up to your account's total concurrency quota, which is 1,000 by default.

Note

For FIFO queues, concurrent invocations are capped either by the number of message group IDs (messageGroupId) or the maximum concurrency setting—whichever is lower.
For example, if you have six message group IDs and maximum concurrency is set to 10, your function can have a maximum of six concurrent invocations.

You can configure maximum concurrency on new and existing Amazon SQS event source mappings.

Configure maximum concurrency using the Lambda console

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. Under Function overview, choose SQS. This opens the Configuration tab.
4. Select the Amazon SQS trigger and choose Edit.
5. For Maximum concurrency, enter a number between 2 and 1,000. To turn off maximum concurrency, leave the box empty.
6. Choose Save.

Configure maximum concurrency using the AWS Command Line Interface (AWS CLI)

Use the `update-event-source-mapping` command with the `--scaling-config` option. Example:

```
aws lambda update-event-source-mapping \
  --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
  --scaling-config '{"MaximumConcurrency":5}'
```

To turn off maximum concurrency, enter an empty value for `--scaling-config`:

```
aws lambda update-event-source-mapping \
  --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
  --scaling-config "{}"
```

Configure maximum concurrency using the Lambda API

Use the `CreateEventSourceMapping` or `UpdateEventSourceMapping` action with a `ScalingConfig` object.

Event source mapping APIs

To manage an event source with the AWS Command Line Interface (AWS CLI) or an AWS SDK, you can use the following API operations:
The following example uses the AWS CLI to map a function named `my-function` to an Amazon SQS queue that is specified by its Amazon Resource Name (ARN), with a batch size of 5 and a batch window of 60 seconds.

```
aws lambda create-event-source-mapping --function-name my-function --batch-size 5 \
--maximum-batching-window-in-seconds 60 \ 
```

You should see the following output:

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 5,
    "MaximumBatchingWindowInSeconds": 60,
    "LastModified": 1541139209.351,
    "State": "Creating",
    "StateTransitionReason": "USER_INITIATED"
}
```

**Backoff strategy for failed invocations**

When an invocation fails, Lambda attempts to retry the invocation while implementing a backoff strategy. The backoff strategy differs slightly depending on whether Lambda encountered the failure due to an error in your function code, or due to throttling.

- **If your function code** caused the error, Lambda gradually backs off retries by reducing the amount of concurrency allocated to your Amazon SQS event source mapping. If invocations continue to fail, Lambda eventually drops the message without retrying.

- **If the invocation fails due to throttling**, Lambda gradually backs off retries by reducing the amount of concurrency allocated to your Amazon SQS event source mapping. Lambda continues...
to retry the message until the message's timestamp exceeds your queue's visibility timeout, at which point Lambda drops the message.

Implementing partial batch responses

When your Lambda function encounters an error while processing a batch, all messages in that batch become visible in the queue again by default, including messages that Lambda processed successfully. As a result, your function can end up processing the same message several times.

To avoid reprocessing successfully processed messages in a failed batch, you can configure your event source mapping to make only the failed messages visible again. This is called a partial batch response. To turn on partial batch responses, specify ReportBatchItemFailures for the FunctionResponseTypes action when configuring your event source mapping. This lets your function return a partial success, which can help reduce the number of unnecessary retries on records.

When ReportBatchItemFailures is activated, Lambda doesn't scale down message polling when function invocations fail. If you expect some messages to fail—and you don't want those failures to impact the message processing rate—use ReportBatchItemFailures.

Note

Keep the following in mind when using partial batch responses:

- If your function throws an exception, the entire batch is considered a complete failure.
- If you're using this feature with a FIFO queue, your function should stop processing messages after the first failure and return all failed and unprocessed messages in batchItemFailures. This helps preserve the ordering of messages in your queue.

To activate partial batch reporting

1. Review the Best practices for implementing partial batch responses.
2. Run the following command to activate ReportBatchItemFailures for your function. To retrieve your event source mapping's UUID, run the `list-event-source-mappings` AWS CLI command.

```bash
aws lambda update-event-source-mapping
```
3. Update your function code to catch all exceptions and return failed messages in a `batchItemFailures` JSON response. The `batchItemFailures` response must include a list of message IDs, as `itemIdentifier` JSON values.

For example, suppose you have a batch of five messages, with message IDs `id1`, `id2`, `id3`, `id4`, and `id5`. Your function successfully processes `id1`, `id3`, and `id5`. To make messages `id2` and `id4` visible again in your queue, your function should return the following response:

```json
{
   "batchItemFailures": [
      {
         "itemIdentifier": "id2"
      },
      {
         "itemIdentifier": "id4"
      }
   ]
}
```

Here are some examples of function code that return the list of failed message IDs in the batch:

**.NET**

**AWS SDK for .NET**

ℹ️ **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](https://github.com/aws-lambda-samples/serverless-snowflake) repository.
// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
[assembly:
]
namespace sqsSample;

public class Function
{
  public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
ILambdaContext context)
  {
    List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
    List<SQSBatchResponse.BatchItemFailure>();
    foreach(var message in evnt.Records)
    {
      try
      {
        //process your message
        await ProcessMessageAsync(message, context);
      }
      catch (System.Exception)
      {
        //Add failed message identifier to the batchItemFailures list
        batchItemFailures.Add(new
        SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
      }
      return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
    ILambdaContext context)
    {
      if (String.IsNullOrEmpty(message.Body))
      {
        throw new Exception("No Body in SQS Message.");
      }
      context.Logger.LogInformation("Processed message {message.Body}");
      // TODO: Do interesting work based on the new message
      await Task.CompletedTask;
    }
}
SDK for Java 2.x

Reporting SQS batch item failures with Lambda using Java.

```java
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context) {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new ArrayList<SQSBatchResponse.BatchItemFailure>();
        String messageId = "";
        for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
            try {
                //process your message
                messageId = message.getMessageId();
            } catch (Exception e) {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.add(new SQSBatchResponse.BatchItemFailure(messageId));
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }
}
```

Implementing partial batch responses
Reporting SQS batch item failures with Lambda using JavaScript.

```javascript
exports.handler = async (event, context) => {

    const batchItemFailures = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ ItemIdentifier: record.messageId });
        }
    }

    return {
        statusCode: 200,
        body: JSON.stringify({ batchItemFailures }),
    };
}

async function processMessageAsync(record, context) {
    if (!record.body) {
        throw new Error('No Body in SQS Message.');
    }
    context.log(`Processed message ${record.body}`);
}
```

### Note
There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.
Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using Python.

```python
import json
def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}

        for record in event['Records']:
            try:
                # process message
            except Exception as e:
                batch_item_failures.append({'itemIdentifier': record['messageId']})

        sqs_batch_response['batchItemFailures'] = batch_item_failures
        return sqs_batch_response
```

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.
AWS Lambda

Developer Guide

Reporting SQS batch item failures with Lambda using Ruby.
require 'json'
def lambda_handler(event:, context:)
if event
batch_item_failures = []
sqs_batch_response = {}
event["Records"].each do |record|
begin
# process message
rescue StandardError => e
batch_item_failures << {"itemIdentifier" => record['messageId']}
end
end
sqs_batch_response["batchItemFailures"] = batch_item_failures
return sqs_batch_response
end
end

Rust
SDK for Rust

Note
There's more on GitHub. Find the complete example and learn how to set up and
run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using Rust.
use aws_lambda_events::{
event::sqs::{SqsBatchResponse, SqsEvent},
sqs::{BatchItemFailure, SqsMessage},
};

Implementing partial batch responses

1348


use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }
    Ok(SqsBatchResponse {
        batch_item_failures,
    })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}

If the failed events do not return to the queue, see [How do I troubleshoot Lambda function SQS ReportBatchItemFailures?](https://aws.amazon.com/knowledge-center/how-to-troubleshoot-aws-lambda-function-sqs-reportbatchitemfailures/) in the AWS Knowledge Center.

**Success and failure conditions**

Lambda treats a batch as a complete success if your function returns any of the following:

- An empty batchItemFailures list
- A null batchItemFailures list
- An empty EventResponse
- A null EventResponse

Implementing partial batch responses
Lambda treats a batch as a complete failure if your function returns any of the following:

- An invalid JSON response
- An empty string `itemIdentifier`
- A null `itemIdentifier`
- An `itemIdentifier` with a bad key name
- An `itemIdentifier` value with a message ID that doesn't exist

**CloudWatch metrics**

To determine whether your function is correctly reporting batch item failures, you can monitor the `NumberOfMessagesDeleted` and `ApproximateAgeOfOldestMessage` Amazon SQS metrics in Amazon CloudWatch.

- `NumberOfMessagesDeleted` tracks the number of messages removed from your queue. If this drops to 0, this is a sign that your function response is not correctly returning failed messages.
- `ApproximateAgeOfOldestMessage` tracks how long the oldest message has stayed in your queue. A sharp increase in this metric can indicate that your function is not correctly returning failed messages.

**Amazon SQS configuration parameters**

All Lambda event source types share the same `CreateEventSourceMapping` and `UpdateEventSourceMapping` API operations. However, only some of the parameters apply to Amazon SQS.

**Event source parameters that apply to Amazon SQS**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required</th>
<th>Default</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BatchSize</td>
<td>N</td>
<td>10</td>
<td>For standard queues, the maximum is 10,000. For FIFO queues, the maximum is 10.</td>
</tr>
<tr>
<td>Enabled</td>
<td>N</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>
## Tutorial: Using Lambda with Amazon SQS

In this tutorial, you create a Lambda function that consumes messages from an Amazon Simple Queue Service (Amazon SQS) queue. The Lambda function runs whenever a new message is added.
to the queue. The function writes the messages to an Amazon CloudWatch Logs stream. The following diagram shows the AWS resources you use to complete the tutorial.

To complete this tutorial, you carry out the following steps:

1. Create a Lambda function that writes messages to CloudWatch Logs.
2. Create an Amazon SQS queue.
3. Create a Lambda event source mapping. The event source mapping reads the Amazon SQS queue and invokes your Lambda function when a new message is added.
4. Test the setup by adding messages to your queue and monitoring the results in CloudWatch Logs.

**Prerequisites**

**Sign up for an AWS account**

If you do not have an AWS account, complete the following steps to create one.

**To sign up for an AWS account**

2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.
When you sign up for an AWS account, an **AWS account root user** is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to an administrative user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to [https://aws.amazon.com/](https://aws.amazon.com/) and choosing **My Account**.

**Create an administrative user**

After you sign up for an AWS account, create an administrative user so that you don’t use the root user for everyday tasks.

**Secure your AWS account root user**

1. Sign in to the [AWS Management Console](https://aws.amazon.com/) as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see [Signing in as the root user](https://aws.amazon.com/) in the **AWS Sign-In User Guide**.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see [Enable a virtual MFA device for your AWS account root user (console)](https://aws.amazon.com/) in the **IAM User Guide**.

**Create an administrative user**

- For your daily administrative tasks, grant administrative access to an administrative user in AWS IAM Identity Center.

  For instructions, see [Getting started](https://aws.amazon.com/) in the **AWS IAM Identity Center User Guide**.

**Sign in as the administrative user**

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.
For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the AWS Sign-In User Guide.

Install the AWS Command Line Interface

If you have not yet installed the AWS Command Line Interface, follow the steps at Installing or updating the latest version of the AWS CLI to install it.

The tutorial requires a command line terminal or shell to run commands. In Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux.

Create the execution role

An execution role is an AWS Identity and Access Management (IAM) role that grants a Lambda function permission to access AWS services and resources. To allow your function to read items from Amazon SQS, attach the AWSLambdaSQSQueueExecutionRole permissions policy.

To create an execution role and attach an Amazon SQS permissions policy

1. Open the Roles page of the IAM console.
2. Choose Create role.
3. For Trusted entity type, choose AWS service.
4. For Use case, choose Lambda.
5. Choose Next.
6. In the Permissions policies search box, enter AWSLambdaSQSQueueExecutionRole.
7. Select the AWSLambdaSQSQueueExecutionRole policy, and then choose Next.
8. Under Role details, for Role name, enter lambda-sqs-role, then choose Create role.

After role creation, note down the Amazon Resource Name (ARN) of your execution role. You'll need it in later steps.

Create the function

Create a Lambda function that processes your Amazon SQS messages. The function code logs the body of the Amazon SQS message to CloudWatch Logs.

This tutorial uses the Node.js 18.x runtime, but we've also provided example code in other runtime languages. You can select the tab in the following box to see code for the runtime you're interested in. The JavaScript code you'll use in this step is in the first example shown in the JavaScript tab.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.
Consuming an SQS event with Lambda using .NET.

```csharp
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
[assembly:
]
namespace SqsIntegrationSampleCode
{
  public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
  {
    foreach (var message in evnt.Records)
    {
      await ProcessMessageAsync(message, context);
    }
    context.Logger.LogInformation("done");
  }

  private async Task ProcessMessageAsync(SQSEvent.SQSMessage message, ILambdaContext context)
  {
    try
    {
      context.Logger.LogInformation($"Processed message {message.Body}");

      // TODO: Do interesting work based on the new message
      await Task.CompletedTask;
    }
    catch (Exception e)
    {
      // You can use Dead Letter Queue to handle failures. By configuring a Lambda DLQ.
      context.Logger.LogError($"An error occurred");
      throw;
    }
  }
}
```
Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using Go.

```
package integration_sqs_to_lambda

import (
    "fmt"
    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
    for _, record := range event.Records {
        err := processMessage(record)
        if err != nil {
            return err
        }
    }
    fmt.Println("done")
    return nil
}

func processMessage(record events.SQSMessage) error {
    fmt.Printf("Processed message %s\n", record.Body)
    // TODO: Do interesting work based on the new message
    return nil
}

func main() {
    lambda.Start(handler)
}
```
Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using Java.

```java
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
            throw e;
        }
    }
}
```
Consuming an SQS event with Lambda using JavaScript.

```javascript
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); // Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Consuming an SQS event with Lambda using TypeScript.

```typescript
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";

export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
```

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.
for (const message of event.Records) {
    await processMessageAsync(message);
}
console.info("done");
}

async function processMessageAsync(message: SQSRecord): Promise<any> {
    try {
        console.log(`Processed message ${message.body}`);
        // TODO: Do interesting work based on the new message
        await Promise.resolve(1); // Placeholder for actual async work
    } catch (err) {
        console.error("An error occurred");
        throw err;
    }
}

Python

SDK for Python (Boto3)

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using Python.

def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
Consuming an SQS event with Lambda using Ruby.

```ruby
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
    rescue StandardError => err
      puts "An error occurred"
      raise err
    end
  end
end
```

To create a Node.js Lambda function

1. Create a directory for the project, and then switch to that directory.
   ```bash
   mkdir sqs-tutorial
   cd sqs-tutorial
   ```

2. Copy the sample JavaScript code into a new file named `index.js`.

3. Create a deployment package using the following `zip` command.
4. Create a Lambda function using the `create-function` AWS CLI command. For the `role` parameter, enter the ARN of the execution role that you created earlier.

```bash
aws lambda create-function --function-name ProcessSQSRecord --zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x --role arn:aws:iam::111122223333:role/lambda-sqs-role
```

## Test the function

1. Test your Lambda function manually using the `invoke` AWS CLI command and a sample Amazon SQS event.

### To invoke the Lambda function with a sample event

1. Save the following JSON as a file named `input.json`. This JSON simulates an event that Amazon SQS might send to your Lambda function, where "body" contains the actual message from the queue. In this example, the message is "test".

   **Example Amazon SQS event**

   This is a test event—you don't need to change the message or the account number.

   ```json
   {
       "Records": [
           {
               "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
           
```
2. Run the following `invoke` AWS CLI command. This command returns CloudWatch logs in the response. For more information about retrieving logs, see [Accessing logs with the AWS CLI](#).

   ```bash
   aws lambda invoke --function-name ProcessSQSRecord --payload file://input.json out --log-type Tail
   --query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --decode
   ```

   The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#) in the [AWS Command Line Interface User Guide for Version 2](#).

3. Find the INFO log in the response. This is where the Lambda function logs the message body. You should see logs that look like this:

   ```
   2023-09-11T22:45:04.271Z 348529ce-2211-4222-9099-59d07d837b60 INFO Processed message test
   2023-09-11T22:45:04.288Z 348529ce-2211-4222-9099-59d07d837b60 INFO done
   ```
Create an Amazon SQS queue that the Lambda function can use as an event source.

**To create a queue**

1. Open the [Amazon SQS console](https://console.aws.amazon.com/sqs/home).  
2. Choose Create queue.  
3. Enter a name for the queue. Leave all other options at the default settings.  
4. Choose Create queue.

After creating the queue, note down its ARN. You need this in the next step when you associate the queue with your Lambda function.

**Configure the event source**
Connect the Amazon SQS queue to your Lambda function by creating an event source mapping. The event source mapping reads the Amazon SQS queue and invokes your Lambda function when a new message is added.

To create a mapping between your Amazon SQS queue and your Lambda function, use the create-event-source-mapping AWS CLI command. Example:

```
aws lambda create-event-source-mapping --function-name ProcessSQSRecord --batch-size 10 \
```

To get a list of your event source mappings, use the list-event-source-mappings command. Example:

```
aws lambda list-event-source-mappings --function-name ProcessSQSRecord
```

**Send a test message**

To send an Amazon SQS message to the Lambda function

1. Open the Amazon SQS console.
2. Choose the queue that you created earlier.
3. Choose Send and receive messages.
4. Under Message body, enter a test message, such as "this is a test message."
5. Choose Send message.
Lambda polls the queue for updates. When there is a new message, Lambda invokes your function with this new event data from the queue. If the function handler returns without exceptions, Lambda considers the message successfully processed and begins reading new messages in the queue. After successfully processing a message, Lambda automatically deletes it from the queue. If the handler throws an exception, Lambda considers the batch of messages not successfully processed, and Lambda invokes the function with the same batch of messages.

**Check the CloudWatch logs**

1. Open the [Functions page](#) of the Lambda console.
2. Choose the [ProcessSQSRecord](#) function.
3. Choose [Monitor](#).
4. Choose [View CloudWatch logs](#).
5. In the CloudWatch console, choose the [Log stream](#) for the function.
6. Find the INFO log. This is where the Lambda function logs the message body. You should see the message that you sent from the Amazon SQS queue. Example:

   2023-09-11T22:49:12.730Z b0c41e9c-0556-5a8b-af83-43e59eefec71 INFO Processed message this is a test message.

**Clean up your resources**

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.
To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose Delete.
4. Enter the name of the role in the text input field and choose Delete.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose Actions, Delete.
4. Type delete in the text input field and choose Delete.

To delete the Amazon SQS queue

1. Sign in to the AWS Management Console and open the Amazon SQS console at https://console.aws.amazon.com/sqs/.
2. Select the queue you created.
3. Choose Delete.
4. Enter confirm in the text input field.
5. Choose Delete.

Tutorial: Using a cross-account Amazon SQS queue as an event source

In this tutorial, you create a Lambda function that consumes messages from an Amazon Simple Queue Service (Amazon SQS) queue in a different AWS account. This tutorial involves two AWS accounts: Account A refers to the account that contains your Lambda function, and Account B refers to the account that contains the Amazon SQS queue.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in Create a Lambda function with the console to create your first Lambda function.
To complete the following steps, you need the AWS Command Line Interface (AWS CLI) version 2. Commands and the expected output are listed in separate blocks:

```bash
aws --version
```

You should see the following output:

```bash
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

⚠️ **Note**

In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system’s built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux. Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

---

## Create the execution role (Account A)

In **Account A**, create an **execution role** that gives your function permission to access the required AWS resources.

**To create an execution role**

1. Open the [Roles page](https://aws.amazon.com) in the AWS Identity and Access Management (IAM) console.
2. Choose **Create role**.
3. Create a role with the following properties.

   - **Trusted entity** – AWS Lambda
   - **Permissions** – `AWSLambdaSQSQueueExecutionRole`
   - **Role name** – `cross-account-lambda-sqs-role`
The AWSLambdaSQSQueueExecutionRole policy has the permissions that the function needs to read items from Amazon SQS and to write logs to Amazon CloudWatch Logs.

Create the function (Account A)

In Account A, create a Lambda function that processes your Amazon SQS messages. The following Node.js 18 code example writes each message to a log in CloudWatch Logs.

Note
For code examples in other languages, see Sample Amazon SQS function code.

Example index.mjs

```javascript
export const handler = async function(event, context) {
    event.Records.forEach(record => {
        const { body } = record;
        console.log(body);
    });
    return {};
}
```

To create the function

Note
Following these steps creates a function in Node.js 18. For other languages, the steps are similar, but some details are different.

1. Save the code example as a file named index.mjs.
2. Create a deployment package.

```bash
zip function.zip index.mjs
```
3. Create the function using the create-function AWS Command Line Interface (AWS CLI) command.

```bash
aws lambda create-function --function-name CrossAccountSQSExample \
```
Test the function (Account A)

In **Account A**, test your Lambda function manually using the `invoke` AWS CLI command and a sample Amazon SQS event.

If the handler returns normally without exceptions, Lambda considers the message to be successfully processed and begins reading new messages in the queue. After successfully processing a message, Lambda automatically deletes it from the queue. If the handler throws an exception, Lambda considers the batch of messages not successfully processed, and Lambda invokes the function with the same batch of messages.

1. Save the following JSON as a file named `input.txt`.

   ```json
   {
     "Records": [
       {
         "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
         "receiptHandle": "AQEBwJnKyrgMgHgX6rYigCgXlS3sLy0a...",
         "body": "test",
         "attributes": {
           "ApproximateReceiveCount": "1",
           "SentTimestamp": "1545082649183",
           "SenderId": "AIDAIENQZJOL023YVJ4V0",
           "ApproximateFirstReceiveTimestamp": "1545082649185"
         },
         "messageAttributes": {},
         "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
         "eventSource": "aws:sqs",
         "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:example-queue",
         "awsRegion": "us-east-1"
       }
     ]
   }
   
   The preceding JSON simulates an event that Amazon SQS might send to your Lambda function, where "body" contains the actual message from the queue.

2. Run the following `invoke` AWS CLI command.
aws lambda invoke --function-name CrossAccountSQSExample \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-aws-configure-set.html) in the *AWS Command Line Interface User Guide for Version 2*.

3. Verify the output in the file `outputfile.txt`.

### Create an Amazon SQS queue (Account B)

In **Account B**, create an Amazon SQS queue that the Lambda function in **Account A** can use as an event source.

**To create a queue**

1. Open the [Amazon SQS console](https://console.aws.amazon.com/sqs).
2. Choose **Create queue**.
3. Create a queue with the following properties.

   - **Type** – **Standard**
   - **Name** – `LambdaCrossAccountQueue`
   - **Configuration** – Keep the default settings.
   - **Access policy** – Choose **Advanced**. Paste in the following JSON policy:

```json
{
    "Version": "2012-10-17",
    "Id": "Queue1_Policy_UUID",
    "Statement": [{
        "Sid": "Queue1_AllActions",
        "Effect": "Allow",
        "Principal": {
            "AWS": [
                "arn:aws:iam::<AccountA_ID>:role/cross-account-lambda-sqs-role"
            ]
        },
        "Action": "sqs:*",
```
This policy grants the Lambda execution role in **Account A** permissions to consume messages from this Amazon SQS queue.

4. After creating the queue, record its Amazon Resource Name (ARN). You need this in the next step when you associate the queue with your Lambda function.

**Configure the event source (Account A)**

In **Account A**, create an event source mapping between the Amazon SQS queue in **Account B** and your Lambda function by running the following `create-event-source-mapping` AWS CLI command.

```bash
aws lambda create-event-source-mapping --function-name CrossAccountSQSExample --batch-size 10 --event-source-arn arn:aws:sqs:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue
```

To get a list of your event source mappings, run the following command.

```bash
aws lambda list-event-source-mappings --function-name CrossAccountSQSExample --event-source-arn arn:aws:sqs:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue
```

**Test the setup**

You can now test the setup as follows:

1. In **Account B**, open the [Amazon SQS console](https://console.aws.amazon.com/sqs).
2. Choose **LambdaCrossAccountQueue**, which you created earlier.
3. Choose **Send and receive messages**.
4. Under **Message body**, enter a test message.
5. Choose **Send message**.

Your Lambda function in **Account A** should receive the message. Lambda will continue to poll the queue for updates. When there is a new message, Lambda invokes your function with this new
event data from the queue. Your function runs and creates logs in Amazon CloudWatch. You can view the logs in the [CloudWatch console](https://console.aws.amazon.com/cloudwatch/home).

**Clean up your resources**

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you’re no longer using, you prevent unnecessary charges to your AWS account.

In **Account A**, clean up your execution role and Lambda function.

**To delete the execution role**

1. Open the [Roles page](https://console.aws.amazon.com/iam/home) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

**To delete the Lambda function**

1. Open the [Functions page](https://console.aws.amazon.com/lambda/home) of the Lambda console.
2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

In **Account B**, clean up the Amazon SQS queue.

**To delete the Amazon SQS queue**

1. Sign in to the AWS Management Console and open the Amazon SQS console at [https://console.aws.amazon.com/sqs/](https://console.aws.amazon.com/sqs/).
2. Select the queue you created.
3. Choose **Delete**.
4. Enter **confirm** in the text input field.
5. Choose **Delete**.
Sample Amazon SQS function code

Sample code is available for the following languages.

Topics

- Node.js
- Java
- C#
- Go
- Python

Node.js

The following is example code that receives an Amazon SQS event message as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Example index.js (Node.js 12)

```javascript
exports.handler = async function(event, context) {
    event.Records.forEach(record => {
        const { body } = record;
        console.log(body);
    });
    return {};
}
```

Zip up the sample code to create a deployment package. For instructions, see Deploy Node.js Lambda functions with .zip file archives.

Java

The following is example Java code that receives an Amazon SQS event message as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

In the code, `handleRequest` is the handler. The handler uses the predefined `SQSEvent` class that is defined in the `aws-lambda-java-events` library.

Example Handler.java

```java
package example;
```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Handler implements RequestHandler<SQSEvent, Void>{
    @Override
    public Void handleRequest(SQSEvent event, Context context) {
        for(SQSMessage msg : event.getRecords()){
            System.out.println(new String(msg.getBody()));
        }
        return null;
    }
}

Dependencies

- aws-lambda-java-core
- aws-lambda-java-events

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see Deploy Java Lambda functions with .zip or JAR file archives.

C#

The following is example C# code that receives an Amazon SQS event message as input and processes it. For illustration, the code writes some of the incoming event data to the console.

In the code, HandleSQSEvent is the handler. The handler uses the predefined SQSEvent class that is defined in the AWS.Lambda.SQSEvents library.

Example ProcessingSQSRecords.cs


namespace SQSLambdaFunction
{
    public class SQSLambdaFunction
    {

Sample code
AWS Lambda

Developer Guide

public string HandleSQSEvent(SQSEvent sqsEvent, ILambdaContext context)
{
Console.WriteLine($"Beginning to process {sqsEvent.Records.Count}
records...");
foreach (var record in sqsEvent.Records)
{
Console.WriteLine($"Message ID: {record.MessageId}");
Console.WriteLine($"Event Source: {record.EventSource}");
Console.WriteLine($"Record Body:");
Console.WriteLine(record.Body);
}
Console.WriteLine("Processing complete.");
return $"Processed {sqsEvent.Records.Count} records.";
}
}
}

Replace the Program.cs in a .NET Core project with the above sample. For instructions, see Build
and deploy C# Lambda functions with .zip ﬁle archives.

Go
The following is example Go code that receives an Amazon SQS event message as input and
processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.
In the code, handler is the handler. The handler uses the predeﬁned SQSEvent class that is
deﬁned in the aws-lambda-go-events library.
Example ProcessSQSRecords.go
package main
import (
"context"
"fmt"
"github.com/aws/aws-lambda-go/events"
"github.com/aws/aws-lambda-go/lambda"
)
Sample code

1376


func handler(ctx context.Context, sqsEvent events.SQSEvent) error {
    for _, message := range sqsEvent.Records {
        fmt.Printf("The message %s for event source %s = %s \n", message.MessageId, message.EventSource, message.Body)
    }
    return nil
}

func main() {
    lambda.Start(handler)
}

Build the executable with `go build` and create a deployment package. For instructions, see [Deploy Go Lambda functions with .zip file archives](#).

**Python**

The following is example Python code that accepts an Amazon SQS record as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

**Example ProcessSQSRecords.py**

```python
from __future__ import print_function

def lambda_handler(event, context):
    for record in event['Records']:
        print("test")
        payload = record["body"]
        print(str(payload))
```

Zip up the sample code to create a deployment package. For instructions, see [Working with .zip file archives for Python Lambda functions](#).

**AWS SAM template for an Amazon SQS application**

You can build this application using [AWS SAM](#). To learn more about creating AWS SAM templates, see [AWS SAM template basics](#) in the [AWS Serverless Application Model Developer Guide](#).

Below is a sample AWS SAM template for the Lambda application from the [tutorial](#). Copy the text below to a .yaml file and save it next to the ZIP package you created previously. Note that the
Handler and Runtime parameter values should match the ones you used when you created the function in the previous section.

**Example template.yaml**

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Example of processing messages on an SQS queue with Lambda
Resources:
  MySQSQueueFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs18.x
    Events:
      MySQSEvent:
        Type: SQS
        Properties:
          Queue: !GetAtt MySqsQueue.Arn
          BatchSize: 10
  MySqsQueue:
    Type: AWS::SQS::Queue
```

For information on how to package and deploy your serverless application using the package and deploy commands, see [Deploying serverless applications](#) in the *AWS Serverless Application Model Developer Guide*. 
Using AWS Lambda with Amazon VPC Lattice

You can register your Lambda functions as targets within an Amazon VPC Lattice service network. By doing this, your Lambda function becomes a service within the network, and clients that have access to the VPC Lattice service network can call your service. If your Lambda function needs to access services within a service network, you can connect your function to a VPC that's already associated with the service network. Having your services live within a VPC Lattice network can help you more easily discover, connect, access, and monitor them.

Topics

- VPC Lattice concepts
- Prerequisites and permissions
- Limitations
- Registering your Lambda function with a VPC Lattice network
- Updating the target of a service in a VPC Lattice network
- Deregistering a Lambda function target
- Cross-account networking
- Receiving events from VPC Lattice
- Sending responses back to VPC Lattice
- Monitoring a service in a VPC Lattice network

VPC Lattice concepts

Throughout this guide, we'll frequently refer to the following VPC Lattice terms:

- **Service** – A service is any software application that can run on an instance, container, or within a serverless function. This topic focuses only on services built using Lambda functions.

- **Service network** – A service network is a logical boundary containing a network of services. This topic covers how to configure your Lambda functions as services within a VPC Lattice service network.

- **Target group** – A target group is a collection of compute type destinations that run a service. Target groups for Lambda can contain only one Lambda function as the target. You cannot have a target group with multiple functions as targets.
• **Listener** – A listener is a process that receives traffic and routes it to different target groups within the service network.

• **Listener rule** – A listener rule encompasses the priority, actions, and conditions that a listener uses to determine where to route traffic. Each listener has a default listener rule, and a listener can have multiple listener rules. A listener rule can contain the following:
  
  • **Protocol** – The protocol that the listener uses to send the request to the destination. Can be either HTTP or HTTPS.
  
  • **Port** – The port that the listener polls for incoming requests. Can be between 1 and 65535 inclusive.
  
  • **Path** – The path to the target resource. For the default listener rule, path is the default path of / . A listener rule can have 6 paths at maximum, including the default path.
  
  • **Priority** – A listener uses the priority of a path to determine which path to route traffic to. A lower number denotes higher priority. The default path has the lowest priority. If you add a new path, VPC Lattice assigns it at the second lowest priority by default. This is just above the priority of the default path, but at a lower priority than all other non-default paths.

In addition, we'll refer to the following AWS Identity and Access Management (IAM) entities:

• **Network owner** – The network owner is the IAM role that owns the VPC Lattice service network.

• **Service owner** – The service owner is the IAM role that owns the service built using a Lambda function. The service owner and the network owner do not have to be the same entity.

### Using Lambda with VPC Lattice and VPC-enabled functions

By default, Lambda functions cannot access any private resources in a VPC. When you configure a Lambda function to connect to private subnets in a VPC, you're allowing the VPC-enabled function to access resources within that VPC. In other words, you're only focusing on the scope of a single VPC.

Lambda functions registered as services in a VPC Lattice network are not the same as VPC-enabled functions, but they can be complementary to each other. When you register a Lambda function as a service in a VPC Lattice network, you're creating an ingress path to your function from one or more VPCs. Additionally, your function may have an optional egress path to another VPC.

When registering a Lambda function as a service, you're focusing on the ingress scenario. This encompasses the specific listener rule configurations that listeners use to route traffic to your
service. From there, your Lambda function may communicate with other AWS services within your VPC through a [Hyperplane ENI (elastic network interface)](hyperplane.eni). The following diagram illustrates this, where the Lambda function is a service within the VPC Lattice network.

![VPC Lattice network diagram]

**Prerequisites and permissions**

This topic assumes that you have both a VPC Lattice service network and a Lambda function. If you don't already have a VPC Lattice service network, refer to the [VPC Lattice user guide to create one](vpc-lattice-guide).

When you register your Lambda function as a target via the Lambda console or AWS Command Line Interface (AWS CLI), Lambda automatically adds the necessary permissions for you.

For Lambda to automatically add permissions for you, so that you can successfully create a Lambda function as a target of a Lambda service, your role must have the following IAM permissions:

- AddPermission
- CreateListener
- CreateService
- CreateServiceNetworkServiceAssociation
To update an existing service in a VPC Lattice network to point to a Lambda function, your role must have the following IAM permissions:

- CreateService
- ListListeners (required for console workflow only)
- ListServices (required for console workflow only)
- RegisterTargets

You can also manually add the permission using the following AWS CLI command:

```
aws lambda add-permission
  --function-name my-function \n  --action lambda:InvokeFunction \n  --statement-id allow-vpc-lattice \n  --principal vpc-lattice.amazonaws.com
  --source-arn target-group-arn
```

Limitations

When using Lambda functions with VPC Lattice networks, keep in mind the following limitations:

- The Lambda function and the target group must be in the same account and in the same Region.
- The maximum size of the request body that you can forward to a service created using a Lambda function is 6 MB.
- The maximum size of the response JSON that the Lambda function can return is 6 MB.
- You can choose HTTP and HTTPS protocols only.

Registering your Lambda function with a VPC Lattice network

You can register any existing Lambda function with a VPC Lattice network using the AWS console or AWS CLI. After registering the function as a service, it can immediately start receiving requests.
To register a Lambda function with a VPC Lattice network (console)

1. Open the Function page of the Lambda console.
2. Choose the name of the function you want to register.
4. In the dropdown menu, select VPC Lattice Application Network.
5. For Intent, choose Create new.
6. For Service name, enter a name for your service.
7. For VPC Lattice network, select the service network that you want to associate this Lambda function with. You can also enter the full Amazon Resource Name (ARN) of the service network. If you don't have an existing service network, you can choose the link in the description, which takes you to the Amazon VPC console where you can create a VPC Lattice network.

   Note

   You don't need to select a VPC Lattice network to finish setting up the trigger. However, if you create the trigger without selecting a network, clients cannot access your service until you associate it with a VPC Lattice network.

8. For Listener, configure the following settings:
   - **Listener name** – Enter a name for your listener
   - **Protocol** – The protocol that the listener uses to send the request to the destination. Can be either HTTP or HTTPS.
   - **Port** – The port that the listener polls for incoming requests. Can be between 1 and 65535 inclusive.
   - Choose Add Listener. VPC Lattice will create a default routing with the default path /. After service creation, you cannot change its name, protocol, and port settings, but you can still define new routing paths.

To register a Lambda function with a VPC Lattice network (AWS CLI)

1. Create a service using the create-service command. Note down the service's ARN in the response. You'll need it in the next steps.
2. Create a target group using the `create-target-group` command. Note down the target group's ARN in the response. You'll need it in the next steps.

```
aws vpc-lattice create-target-group
  --name my-vpc-lattice-target-group
  --type LAMBDA
```

**Note**

By default, this command creates a target group that sends events with the **V1 event structure** to your Lambda function. To send a **V2 event structure** that's compatible with Amazon API Gateway, specify the `LambdaEventStructureVersion` option. For example, append `--lambda-event-structure-version V2` to the end of the previous command.

3. Create a listener within the service network using the `create-listener` command. New listeners automatically use the default path of `/`. Replace the value of the `service` parameter with your service's ARN from step 1. Replace the value of `TargetGroupArn` in the `default-action` parameter with your target group ARN from step 2.

```
aws vpc-lattice create-listener
  --name https
  --service-identifier svc-0e2f2665e1cebb720
  --protocol HTTPS
  --default-action
  forward='{targetGroups={targetGroupIdentifier=tg-0e2f2665e1cebb720}}'
```

4. Register your Lambda function as a target using the `register-targets` command. Replace the value of the `target-group-arn` parameter with your target group ARN from step 2. Replace the value of `Id` in the `targets` parameter with the ARN of your Lambda function.

```
aws vpc-lattice register-targets
  --target-group-identifier arn:aws:vpc-lattice:us-west-2:123456789012:targetgroup/tg-0e2f2665e1cebb720
  --targets id=arn:aws:lambda:us-west-2:123456789012:function:my-function
```
Note

In the previous register-targets command, if your Lambda function doesn't already explicitly allow VPC Lattice to invoke it, VPC Lattice attaches the necessary permissions to your function's execution role automatically. To allow VPC Lattice to automatically attach permissions, your role needs to have the AddPermission permission.

5. Associate the service with the service network using the create-service-network-service-association command. Replace the value of the service parameter with your service ARN from step 1. Replace the value of the service-network parameter with the ARN of your VPC Lattice service network.

```bash
aws vpc-lattice create-service-network-service-association \
  --service-identifier arn:aws:vpc-lattice:us-west-2:123456789012:service/svc-0b9b89d907bc8668c \
  --service-network-identifier arn:aws:vpc-lattice:us-west-2:123456789012:servicenetwork/03d622a31e5154247
```

### Updating the target of a service in a VPC Lattice network

You can update any existing service within a VPC Lattice network to point to a Lambda function target. You can do this by adding a new routing path using the AWS console or AWS CLI. When you add a new path, VPC Lattice assigns that path the second lowest priority, just above the default path (lowest priority).

**To update the target of a service to point to a Lambda function (console)**

1. Open the Function page of the Lambda console.
2. Choose the name of the function you want to register.
4. In the dropdown menu, select VPC Lattice Application Network.
5. For Intent, choose Select existing.
6. For Service name, choose an existing service.
7. For Listener, choose an existing listener.
8. For **Rule name**, enter a name for the new rule.

9. For **Path**, define a new routing path for the listener.

10. Choose **Add**.

For the following steps, you'll need the ARN of your service, as well as the ARN of the listener that you want to add a new rule for.

1. Create a new target group using the `create-target-group` command. Note down the target group's ARN in the response, as you'll need it for future steps.

   ```bash
   aws vpc-lattice create-target-group  
   --name my-vpc-lattice-target-group  
   --type LAMBDA
   ```

2. Create a new rule for your existing listener using the `create-rule` command. This command assumes that your conditions are in a file called `conditions-pattern.json` in your current directory. Replace the value of the `listener-arn` parameter with the ARN of your listener. Replace the value of `TargetGroupArn` in the `actions` parameter with your target group ARN from step 1.

   ```bash
   aws vpc-lattice create-rule 
   --name my-rule  
   --priority 1  
   --listener-identifier listener-0e9af499f72e5251b  
   --service-identifier svc-01755f67d3a427803  
   --match httpMatch='{pathMatch={match={prefix="/test"}}}'  
   --default action forward='{targetGroups=[{targetGroupIdentifier=tg-042d5b70f1e743940}]}'
   ```

3. Register your Lambda function as a target using the `register-targets` command. Replace the value of the `target-group-arn` parameter with your target group ARN from step 2. Replace the value of `Id` in the `targets` parameter with the ARN of your Lambda function.

   ```bash
   aws vpc-lattice register-targets 
   --target-group-identifier arn:aws:vpc-lattice:us-west-2:123456789012:targetgroup/tg-0e2f2665e1cebb720 
   ```
**Deregistering a Lambda function target**

To deregister a Lambda function target in a VPC Lattice network, use the VPC console. For more information, see the VPC Lattice user guide.

Alternatively, you can use the following AWS CLI command:

```bash
aws vpc-lattice deregister-targets \
    --target-group-identifier arn:aws:vpc-lattice:us-west-2:123456789012:targetgroup/tg-0e2f2665e1ceb720 \
```

You cannot deregister a service built using Lambda functions from the Lambda console.

**Cross-account networking**

Services within your VPC Lattice service network don't have to all be in the same AWS account. In addition, the VPC Lattice service network itself can reside in a different account. This means that you can associate a service built using Lambda functions with a service network in a different AWS account. You'll need specific permissions from the network owner to create these service associations. For more information about permissions required, see [Prepare to call the VPC Lattice API](#) in the Amazon VPC Lattice user guide.

You can create an association between a Lambda function with a service network in a different account through the AWS console. To do this, instead of choosing the **VPC Lattice network** from the dropdown menu (this only displays networks in your account), paste the full ARN of the network.

In general, you can create an association between any service and any service network with the `create-service-network-service-association` AWS CLI command. This means you can manage your service networks in a central account and have services built using Lambda functions in other accounts across your AWS organization. In the following example, note that the service and service-network live in two different accounts:

```bash
aws vpc-lattice create-service-network-service-association \
    --service arn:aws:vpc-lattice:us-west-2:123456789012:service/svc-0b9b89d907bc8668c \
        --service-network arn:aws:vpc-lattice:us-west-2:444455556666:servicenetwork/03d622a31e5154247
```
Receiving events from VPC Lattice

The VPC Lattice service routes Lambda invocation requests over HTTP and HTTPS. The event structure version that your function receives depends on your LambdaEventStructureVersion setting when you created your target group.

Sample V1 event structure

The following is an example V1 event that your Lambda function might receive from VPC Lattice, in JSON format:

```json
{
    "raw_path": "/path/to/resource",
    "method": "GET|POST|HEAD|...",
    "headers": {
        "header-key": "header-value",
        ... },
    "query_string_parameters": {
        "key": "value",
        ... },
    "body": "request-body",
    "is_base64_encoded": true|false
}
```

If the content-encoding header is not present, Base64 encoding depends on the content type. For the content types, text/*, application/json, application/xml, and application/javascript, the service sends the body as is and sets isBase64Encoded to false.

⚠️ Note

The request body can have a maximum size of 1023 KiB if sent in plaintext, or 767 KiB if base64 encoded. The list of request headers can have a maximum of 50 key-value pairs.

Sample V2 event structure

The following is an example V2 event that your Lambda function might receive from VPC Lattice, in JSON format:

```json
{
    "version": "2.0",
    "path": "/",
    "method": "GET|POST|HEAD|...",
    ...
}
```
"requestContext": { "serviceNameArn": "arn:aws:vpc-lattice:region:123456789012:service/svc-0a40eebed65f8d69c", "serviceArn": "arn:aws:vpc-lattice:region:123456789012:service/vpc-0b8276c84697e7339", "targetGroupArn": "arn:aws:vpc-lattice:region:123456789012:targetgroup/tg-6d0ecf831ee9f09", "identity": { "sourceVpcArn": "arn:aws:ec2:region:123456789012:vpc/vpc-0b8276c84697e7339", "principal": "arn:aws:sts::123456789012:assumed-role/example-role/057d00f8b51257ba3c853a0f248943cf", "sessionName": "057d00f8b51257ba3c853a0f248943cf", "x509SanDns": "example.com" }, "region": "region", "timeEpoch": "1690497599177430" }

Sending responses back to VPC Lattice

When you send a response from your Lambda function back to VPC Lattice, the response must include the Base64 encoding status, status code, and relevant headers. The body is optional. The following is an example response in JSON format:

```json
{
  "isBase64Encoded": false,
  "statusCode": 200,
  "statusDescription": "200 OK",
  "headers": {
    "Set-Cookie": "cookies",
    "Content-Type": "application/json"
  },
  "body": "Hello from Lambda (optional)"
}
```

To include binary content in the response body, you must Base64 encode the content and set `isBase64Encoded` to `true`. This tells VPC Lattice to decode the content before sending the response to the client.
VPC Lattice doesn't honor hop-by-hop headers such as Connection or Transfer-Encoding. Also, you can omit the Content-Length header because VPC Lattice automatically computes it before sending responses to clients.

Note

The response body can have a maximum size of 1023 KiB if sent in plaintext, or 767 KiB if base64 encoded. The list of response headers can have a maximum of 50 key-value pairs.

Monitoring a service in a VPC Lattice network

To monitor services built using Lambda functions in a VPC Lattice network, VPC Lattice provides Amazon CloudWatch metrics, AWS CloudTrail logs, and access logs. These tools can help you track key performance metrics such as the total number of requests to your service and the number of connection timeouts.

By default, Lambda automatically emits metrics to CloudWatch and event history logs to CloudTrail. Access logs are optional, and Lambda deactivates them by default. For more information on monitoring, see Monitoring Amazon VPC Lattice in the VPC Lattice user guide.
Best practices for working with AWS Lambda functions

The following are recommended best practices for using AWS Lambda:

Topics

• Function code
• Function configuration
• Metrics and alarms
• Working with streams
• Security best practices

For more information about best practices for Lambda applications, see Application design in Serverless Land.

Function code

• Separate the Lambda handler from your core logic. This allows you to make a more unit-testable function. In Node.js this may look like:

```javascript
exports.myHandler = function(event, context, callback) {
    var foo = event.foo;
    var bar = event.bar;
    var result = MyLambdaFunction (foo, bar);

    callback(null, result);
}

function MyLambdaFunction (foo, bar) {
    // MyLambdaFunction logic here
}
```

• Take advantage of execution environment reuse to improve the performance of your function. Initialize SDK clients and database connections outside of the function handler, and cache static assets locally in the /tmp directory. Subsequent invocations processed by the same instance of your function can reuse these resources. This saves cost by reducing function run time.
To avoid potential data leaks across invocations, don’t use the execution environment to store user data, events, or other information with security implications. If your function relies on a mutable state that can’t be stored in memory within the handler, consider creating a separate function or separate versions of a function for each user.

- **Use a keep-alive directive to maintain persistent connections.** Lambda purges idle connections over time. Attempting to reuse an idle connection when invoking a function will result in a connection error. To maintain your persistent connection, use the keep-alive directive associated with your runtime. For an example, see [Reusing Connections with Keep-Alive in Node.js](#).

- **Use environment variables to pass operational parameters to your function.** For example, if you are writing to an Amazon S3 bucket, instead of hard-coding the bucket name you are writing to, configure the bucket name as an environment variable.

- **Control the dependencies in your function's deployment package.** The AWS Lambda execution environment contains a number of libraries such as the AWS SDK for the Node.js and Python runtimes (a full list can be found here: [Lambda runtimes](#)). To enable the latest set of features and security updates, Lambda will periodically update these libraries. These updates may introduce subtle changes to the behavior of your Lambda function. To have full control of the dependencies your function uses, package all of your dependencies with your deployment package.

- **Minimize your deployment package size to its runtime necessities.** This will reduce the amount of time that it takes for your deployment package to be downloaded and unpacked ahead of invocation. For functions authored in Java or .NET Core, avoid uploading the entire AWS SDK library as part of your deployment package. Instead, selectively depend on the modules which pick up components of the SDK you need (e.g. DynamoDB, Amazon S3 SDK modules and [Lambda core libraries](#)).

- **Reduce the time it takes Lambda to unpack deployment packages** authored in Java by putting your dependency `.jar` files in a separate `/lib` directory. This is faster than putting all your function's code in a single jar with a large number of `.class` files. See [Deploy Java Lambda functions with .zip or JAR file archives](#) for instructions.

- **Minimize the complexity of your dependencies.** Prefer simpler frameworks that load quickly on execution environment startup. For example, prefer simpler Java dependency injection (IoC) frameworks like Dagger or Guice, over more complex ones like Spring Framework.

- **Avoid using recursive code** in your Lambda function, wherein the function automatically calls itself until some arbitrary criteria is met. This could lead to unintended volume of function invocations and escalated costs. If you do accidentally do so, set the function reserved
concurrency to 0 immediately to throttle all invocations to the function, while you update the code.

- **Do not use non-documented, non-public APIs** in your Lambda function code. For AWS Lambda managed runtimes, Lambda periodically applies security and functional updates to Lambda's internal APIs. These internal API updates may be backwards-incompatible, leading to unintended consequences such as invocation failures if your function has a dependency on these non-public APIs. See the API reference for a list of publicly available APIs.

- **Write idempotent code.** Writing idempotent code for your functions ensures that duplicate events are handled the same way. Your code should properly validate events and gracefully handle duplicate events. For more information, see How do I make my Lambda function idempotent?.

## Function configuration

- **Performance testing your Lambda function** is a crucial part in ensuring you pick the optimum memory size configuration. Any increase in memory size triggers an equivalent increase in CPU available to your function. The memory usage for your function is determined per-invocation and can be viewed in Amazon CloudWatch. On each invoke a REPORT: entry will be made, as shown below:

  ```
  REPORT RequestId: 3604209a-e9a3-11e6-939a-754dd98c7be3 Duration: 12.34 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 18 MB
  ```

  By analyzing the Max Memory Used: field, you can determine if your function needs more memory or if you over-provisioned your function's memory size.

  **To find the right memory configuration** for your functions, we recommend using the open source AWS Lambda Power Tuning project. For more information, see AWS Lambda Power Tuning on GitHub.

  To optimize function performance, we also recommend deploying libraries that can leverage Advanced Vector Extensions 2 (AVX2). This allows you to process demanding workloads, including machine learning inferencing, media processing, high performance computing (HPC), scientific simulations, and financial modeling. For more information, see Creating faster AWS Lambda functions with AVX2.
• **Load test your Lambda function** to determine an optimum timeout value. It is important to analyze how long your function runs so that you can better determine any problems with a dependency service that may increase the concurrency of the function beyond what you expect. This is especially important when your Lambda function makes network calls to resources that may not handle Lambda's scaling.

• **Use most-restrictive permissions when setting IAM policies.** Understand the resources and operations your Lambda function needs, and limit the execution role to these permissions. For more information, see [Lambda resource access permissions](https://docs.aws.amazon.com/lambda/latest/dg/permissions.html).

• **Be familiar with Lambda quotas.** Payload size, file descriptors and /tmp space are often overlooked when determining runtime resource limits.

• **Be familiar with your upstream and downstream throughput constraints.** While Lambda functions scale seamlessly with load, upstream and downstream dependencies may not have the same throughput capabilities. If you need to limit how high your function can scale, configure reserved concurrency.

• **Delete Lambda functions that you are no longer using.** By doing so, the unused functions won't needlessly count against your deployment package size limit.

• **If you are using Amazon Simple Queue Service** as an event source, make sure the value of the function's expected invocation time does not exceed the Visibility Timeout value on the queue. This applies both to CreateFunction and UpdateFunctionConfiguration.

  • In the case of CreateFunction, AWS Lambda will fail the function creation process.

  • In the case of UpdateFunctionConfiguration, it could result in duplicate invocations of the function.

**Metrics and alarms**

• **Use Working with Lambda function metrics** and **CloudWatch Alarms** instead of creating or updating a metric from within your Lambda function code. It's a much more efficient way to track the health of your Lambda functions, allowing you to catch issues early in the development process. For instance, you can configure an alarm based on the expected duration of your Lambda function invocation in order to address any bottlenecks or latencies attributable to your function code.

• **Leverage your logging library and AWS Lambda Metrics and Dimensions** to catch app errors (e.g. ERR, ERROR, WARNING, etc.)
• **Use AWS Cost Anomaly Detection** to detect unusual activity on your account. Cost Anomaly Detection uses machine learning to continuously monitor your cost and usage while minimizing false positive alerts. Cost Anomaly Detection uses data from AWS Cost Explorer, which has a delay of up to 24 hours. As a result, it can take up to 24 hours to detect an anomaly after usage occurs. To get started with Cost Anomaly Detection, you must first [sign up for Cost Explorer]. Then, [access Cost Anomaly Detection].

**Working with streams**

• **Test with different batch and record sizes** so that the polling frequency of each event source is tuned to how quickly your function is able to complete its task. The CreateEventSourceMapping BatchSize parameter controls the maximum number of records that can be sent to your function with each invoke. A larger batch size can often more efficiently absorb the invoke overhead across a larger set of records, increasing your throughput.

By default, Lambda invokes your function as soon as records are available. If the batch that Lambda reads from the event source has only one record in it, Lambda sends only one record to the function. To avoid invoking the function with a small number of records, you can tell the event source to buffer records for up to 5 minutes by configuring a batching window. Before invoking the function, Lambda continues to read records from the event source until it has gathered a full batch, the batching window expires, or the batch reaches the payload limit of 6 MB. For more information, see [Batching behavior].

• **Increase Kinesis stream processing throughput by adding shards.** A Kinesis stream is composed of one or more shards. Lambda will poll each shard with at most one concurrent invocation. For example, if your stream has 100 active shards, there will be at most 100 Lambda function invocations running concurrently. Increasing the number of shards will directly increase the number of maximum concurrent Lambda function invocations and can increase your Kinesis stream processing throughput. If you are increasing the number of shards in a Kinesis stream, make sure you have picked a good partition key (see [Partition Keys]) for your data, so that related records end up on the same shards and your data is well distributed.

• **Use Amazon CloudWatch** on IteratorAge to determine if your Kinesis stream is being processed. For example, configure a CloudWatch alarm with a maximum setting to 30000 (30 seconds).
Security best practices

Lambda resource access permissions

You can use AWS Identity and Access Management (IAM) to manage access to the Lambda API and resources such as functions and layers. For users and applications in your account that use Lambda, you can create IAM policies that apply to users, groups, or roles.

Every Lambda function has an IAM role called an execution role. In this role, you can attach a policy that defines the permissions that your function needs to access other AWS services and resources. At a minimum, your function needs access to Amazon CloudWatch Logs for log streaming. If your function calls other service APIs with the AWS SDK, you must include the necessary permissions in the execution role's policy. Lambda also uses the execution role to get permission to read from event sources when you use an event source mapping to invoke your function.

To give other accounts and AWS services permission to use your Lambda resources, use a resource-based policy. Lambda resources include functions, versions, aliases, and layer versions. When a user tries to access a Lambda resource, Lambda considers both the user's identity-based policies and the resource's resource-based policy. When an AWS service such as Amazon Simple Storage Service (Amazon S3) calls your Lambda function, Lambda considers only the resource-based policy.

To manage permissions for users and applications in your account, we recommend using an AWS managed policy. You can use these managed policies as-is, or as a starting point for writing your own more restrictive policies. Policies can restrict user permissions by the resource that an action affects, and by additional optional conditions. For more information, see Resources and conditions for Lambda actions.

If your Lambda functions contain calls to other AWS resources, you might also want to restrict which functions can access those resources. To do this, include the lambda:SourceFunctionArn condition key in an IAM identity-based policy or service control policy (SCP) for the target resource. For more information, see Working with Lambda execution environment credentials.

For more information about IAM, see the IAM User Guide.

For more information about applying security principles to Lambda applications, see Security in Serverless Land.

Topics

- Lambda execution role
- Identity-based IAM policies for Lambda
• Attribute-based access control for Lambda
• Using resource-based policies for Lambda
• Resources and conditions for Lambda actions
• Using permissions boundaries for AWS Lambda applications
Lambda execution role

A Lambda function's execution role is an AWS Identity and Access Management (IAM) role that grants the function permission to access AWS services and resources. For example, you might create an execution role that has permission to send logs to Amazon CloudWatch and upload trace data to AWS X-Ray. This page provides information on how to create, view, and manage a Lambda function's execution role.

You provide an execution role when you create a function. When you invoke your function, Lambda automatically provides your function with temporary credentials by assuming this role. You don't have to call sts:AssumeRole in your function code.

In order for Lambda to properly assume your execution role, the role's trust policy must specify the Lambda service principal (lambda.amazonaws.com) as a trusted service.

To view a function's execution role

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. Choose Configuration, and then choose Permissions.
4. Under Resource summary, review the services and resources that the function can access.
5. Choose a service from the dropdown list to see permissions related to that service.

You can add or remove permissions from a function's execution role at any time, or configure your function to use a different role. Add permissions for any services that your function calls with the AWS SDK, and for services that Lambda uses to enable optional features.

When you add permissions to your function, update its code or configuration as well. This forces running instances of your function, which have outdated credentials, to stop and be replaced.

Topics

- Creating an execution role in the IAM console
- Grant least privilege access to your Lambda execution role
- Managing roles with the IAM API
- Session duration for temporary security credentials
- AWS managed policies for Lambda features
Creating an execution role in the IAM console

By default, Lambda creates an execution role with minimal permissions when you create a function in the Lambda console. You can also create an execution role in the IAM console.

To create an execution role in the IAM console

1. Open the Roles page in the IAM console.
2. Choose Create role.
3. Under Use case, choose Lambda.
4. Choose Next.
5. Select the AWS managed policies AWSLambdaBasicExecutionRole and AWSXRayDaemonWriteAccess.
6. Choose Next.
7. Enter a Role name and then choose Create role.

For detailed instructions, see Creating a role for an AWS service (console) in the IAM User Guide.

Grant least privilege access to your Lambda execution role

When you first create an IAM role for your Lambda function during the development phase, you might sometimes grant permissions beyond what is required. Before publishing your function in the production environment, as a best practice, adjust the policy to include only the required permissions. For more information, see Apply least-privilege permissions in the IAM User Guide.

Use IAM Access Analyzer to help identify the required permissions for the IAM execution role policy. IAM Access Analyzer reviews your AWS CloudTrail logs over the date range that you specify and generates a policy template with only the permissions that the function used during that time. You can use the template to create a managed policy with fine-grained permissions, and then attach it to the IAM role. That way, you grant only the permissions that the role needs to interact with AWS resources for your specific use case.

For more information, see Generate policies based on access activity in the IAM User Guide.
Managing roles with the IAM API

To create an execution role with the AWS Command Line Interface (AWS CLI), use the `create-role` command. When using this command, you can specify the trust policy inline. A role's trust policy gives the specified principals permission to assume the role. In the following example, you grant the Lambda service principal permission to assume your role. Note that requirements for escaping quotes in the JSON string may vary depending on your shell.

```
aws iam create-role --role-name lambda-ex --assume-role-policy-document '{
"Version": "2012-10-17","Statement": [{ "Effect": "Allow", "Principal": {"Service": "lambda.amazonaws.com"}, "Action": "sts:AssumeRole"}]
}
```

You can also define the trust policy for the role using a separate JSON file. In the following example, `trust-policy.json` is a file in the current directory.

**Example trust-policy.json**

```
{
    "Version": "2012-10-17",
    "Statement": [
        { "Effect": "Allow",
          "Principal": {
            "Service": "lambda.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
    ]
}
```

```
aws iam create-role --role-name lambda-ex --assume-role-policy-document file://trust-policy.json
```

You should see the following output:

```
{
    "Role": {
        "Path": "/",
        "RoleName": "lambda-ex",
        "RoleId": "AROAQFOXMP6TZ6ITKWNND",
        ...
    }
}
```
"Arn": "arn:aws:iam::123456789012:role/lambda-ex",
"CreateDate": "2020-01-17T23:19:12Z",
"AssumeRolePolicyDocument": {
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}

Note
Lambda automatically assumes your execution role when you invoke your function. You should avoid calling `sts:AssumeRole` manually in your function code. If your use case requires that the role assumes itself, you must include the role itself as a trusted principal in your role's trust policy. For more information on how to modify a role trust policy, see Modifying a role trust policy (console) in the IAM User Guide.

To add permissions to the role, use the `attach-policy-to-role` command. Start by adding the AWSLambdaBasicExecutionRole managed policy.

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

Session duration for temporary security credentials

Lambda assumes the execution role associated with your function to fetch temporary security credentials which are then available as environment variables during a function's invocation. If you use these temporary credentials outside of Lambda, such as to create a presigned Amazon S3 URL, you can't control the session duration. The IAM maximum session duration setting doesn't apply to sessions that are assumed by AWS services such as Lambda. Use the `sts:AssumeRole` action if you need control over session duration.
# AWS managed policies for Lambda features

The following AWS managed policies provide permissions that are required to use Lambda features.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AWSLambdaMSKExecutionRole</strong> – Lambda added the <code>kafka:DescribeClusterV2</code> permission to this policy.</td>
<td>AWSLambdaMSKExecutionRole grants permissions to read and access records from an Amazon Managed Streaming for Apache Kafka (Amazon MSK) cluster, manage elastic network interfaces (ENIs), and write to CloudWatch Logs.</td>
<td>June 17, 2022</td>
</tr>
<tr>
<td><strong>AWSLambdaBasicExecutionRole</strong> – Lambda started tracking changes to this policy.</td>
<td>AWSLambdaBasicExecutionRole grants permissions to upload logs to CloudWatch.</td>
<td>February 14, 2022</td>
</tr>
<tr>
<td><strong>AWSLambdaDynamoDBExecutionRole</strong> – Lambda started tracking changes to this policy.</td>
<td>AWSLambdaDynamoDBExecutionRole grants permissions to read records from an Amazon DynamoDB stream and write to CloudWatch Logs.</td>
<td>February 14, 2022</td>
</tr>
<tr>
<td><strong>AWSLambdaKinesisExecutionRole</strong> – Lambda started tracking changes to this policy.</td>
<td>AWSLambdaKinesisExecutionRole grants permissions to read events from an Amazon Kinesis data stream and write to CloudWatch Logs.</td>
<td>February 14, 2022</td>
</tr>
<tr>
<td><strong>AWSLambdaMSKExecutionRole</strong> – Lambda started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>-------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>tracking changes to this policy.</td>
<td>ns to read and access records from an Amazon Managed Streaming for Apache Kafka (Amazon MSK) cluster, manage elastic network interfaces (ENIs), and write to CloudWatch Logs.</td>
<td></td>
</tr>
<tr>
<td><strong>AWSLambdaSQSQueueExecutionRole</strong> – Lambda</td>
<td><strong>AWSLambdaSQSQueueExecutionRole</strong> grants permissions to read a message from an Amazon Simple Queue Service (Amazon SQS) queue and write to CloudWatch Logs.</td>
<td>February 14, 2022</td>
</tr>
<tr>
<td><strong>AWSLambdaVPCAccessExecutionRole</strong> – Lambda</td>
<td><strong>AWSLambdaVPCAccessExecutionRole</strong> grants permissions to manage ENIs within an Amazon VPC and write to CloudWatch Logs.</td>
<td>February 14, 2022</td>
</tr>
<tr>
<td><strong>AWSXRayDaemonWriteAccess</strong> – Lambda</td>
<td><strong>AWSXRayDaemonWriteAccess</strong> grants permissions to upload trace data to X-Ray.</td>
<td>February 14, 2022</td>
</tr>
<tr>
<td><strong>CloudWatchLambdaInsightsExecutionRolePolicy</strong> – Lambda</td>
<td><strong>CloudWatchLambdaInsightsExecutionRolePolicy</strong> grants permissions to write runtime metrics to CloudWatch Lambda Insights.</td>
<td>February 14, 2022</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>AmazonS3ObjectLambdaExecutionRolePolicy – Lambda started tracking changes to this policy.</td>
<td>AmazonS3ObjectLambdaExecutionRolePolicy grants permissions to interact with Amazon Simple Storage Service (Amazon S3) object Lambda and to write to CloudWatch Logs.</td>
<td>February 14, 2022</td>
</tr>
</tbody>
</table>

For some features, the Lambda console attempts to add missing permissions to your execution role in a customer managed policy. These policies can become numerous. To avoid creating extra policies, add the relevant AWS managed policies to your execution role before enabling features.

When you use an event source mapping to invoke your function, Lambda uses the execution role to read event data. For example, an event source mapping for Kinesis reads events from a data stream and sends them to your function in batches.

When a service assumes a role in your account, you can include the aws:SourceAccount and aws:SourceArn global condition context keys in your role trust policy to limit access to the role to only requests that are generated by expected resources. For more information, see [Cross-service confused deputy prevention for AWS Security Token Service](#).

You can use event source mappings with the following services:

**Services that Lambda reads events from**

- [Amazon DynamoDB](#)
- [Amazon Kinesis](#)
- [Amazon MQ](#)
- [Amazon Managed Streaming for Apache Kafka (Amazon MSK)](#)
- [Self-managed Apache Kafka](#)
- [Amazon Simple Queue Service (Amazon SQS)](#)
- [Amazon DocumentDB (with MongoDB compatibility) (Amazon DocumentDB)](#)

AWS managed policies for Lambda features
In addition to the AWS managed policies, the Lambda console provides templates for creating a custom policy with permissions for additional use cases. When you create a function in the Lambda console, you can choose to create a new execution role with permissions from one or more templates. These templates are also applied automatically when you create a function from a blueprint, or when you configure options that require access to other services. Example templates are available in this guide’s GitHub repository.

Working with Lambda execution environment credentials

It's common for your Lambda function code to make API requests to other AWS services. To make these requests, Lambda generates an ephemeral set of credentials by assuming your function's execution role. These credentials are available as environment variables during your function's invocation. When working with AWS SDKs, you don’t need to provide credentials for the SDK directly in code. By default, the credential provider chain sequentially checks each place where you can set credentials and selects the first one available—usually the environment variables (AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, and AWS_SESSION_TOKEN).

Lambda injects the source function ARN into the credentials context if the request is an AWS API request that comes from within your execution environment. Lambda also injects the source function ARN for the following AWS API requests that Lambda makes on your behalf outside of your execution environment:

<table>
<thead>
<tr>
<th>Service</th>
<th>Action</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloudWatch Logs</td>
<td>CreateLogGroup, CreateLogStream, PutLogEvents</td>
<td>To store logs into a CloudWatch Logs log group</td>
</tr>
<tr>
<td>X-Ray</td>
<td>PutTraceSegments</td>
<td>To send trace data to X-Ray</td>
</tr>
<tr>
<td>Amazon EFS</td>
<td>ClientMount</td>
<td>To connect your function to an Amazon Elastic File System (Amazon EFS) file system</td>
</tr>
</tbody>
</table>

Other AWS API calls that Lambda makes outside of your execution environment on your behalf using the same execution role don't contain the source function ARN. Examples of such API calls outside the execution environment include:
• Calls to AWS Key Management Service (AWS KMS) to automatically encrypt and decrypt your environment variables.

• Calls to Amazon Elastic Compute Cloud (Amazon EC2) to create elastic network interfaces (ENIs) for a VPC-enabled function.

• Calls to AWS services, such as Amazon Simple Queue Service (Amazon SQS), to read from an event source that's set up as an event source mapping.

With the source function ARN in the credentials context, you can verify whether a call to your resource came from a specific Lambda function's code. To verify this, use the `lambda:SourceFunctionArn` condition key in an IAM identity-based policy or service control policy (SCP).

⚠️ Note

You cannot use the `lambda:SourceFunctionArn` condition key in resource-based policies.

With this condition key in your identity-based policies or SCPs, you can implement security controls for the API actions that your function code makes to other AWS services. This has a few key security applications, such as helping you identify the source of a credential leak.

⚠️ Note

The `lambda:SourceFunctionArn` condition key is different from the `lambda:FunctionArn` and `aws:SourceArn` condition keys. The `lambda:FunctionArn` condition key applies only to event source mappings and helps define which functions your event source can invoke. The `aws:SourceArn` condition key applies only to policies where your Lambda function is the target resource, and helps define which other AWS services and resources can invoke that function. The `lambda:SourceFunctionArn` condition key can apply to any identity-based policy or SCP to define the specific Lambda functions that have permissions to make specific AWS API calls to other resources.

To use `lambda:SourceFunctionArn` in your policy, include it as a condition with any of the ARN condition operators. The value of the key must be a valid ARN.
For example, suppose your Lambda function code makes an `s3:PutObject` call that targets a specific Amazon S3 bucket. You might want to allow only one specific Lambda function to have `s3:PutObject` access to that bucket. In this case, your function's execution role should have a policy attached that looks like this:

**Example policy granting a specific Lambda function access to an Amazon S3 resource**

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ExampleSourceFunctionArn",
            "Effect": "Allow",
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::lambda_bucket/**",
            "Condition": {
                "ArnEquals": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:source_lambda"
                }
            }
        }
    ]
}
```

This policy allows only `s3:PutObject` access if the source is the Lambda function with ARN `arn:aws:lambda:us-east-1:123456789012:function:source_lambda`. This policy doesn't allow `s3:PutObject` access to any other calling identity. This is true even if a different function or entity makes an `s3:PutObject` call with the same execution role.

**Note**

The `lambda:SourceFunctionARN` condition key doesn't support Lambda function versions or function aliases. If you use the ARN for a particular function version or alias, your function won't have permission to take the action you specify. Be sure to use the unqualified ARN for your function without a version or alias suffix.
You can also use `lambda:SourceFunctionArn` in service control policies. For example, suppose you want to restrict access to your bucket to either a single Lambda function's code or to calls from a specific Amazon Virtual Private Cloud (VPC). The following SCP illustrates this.

**Example policy denying access to Amazon S3 under specific conditions**

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": ["s3:*"],
            "Resource": "arn:aws:s3:::lambda_bucket/**",
            "Effect": "Deny",
            "Condition": {
                "StringNotEqualsIfExists": {
                    "aws:SourceVpc": [
                        "vpc-12345678"
                    ]
                },
                "ArnNotEqualsIfExists": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:source_lambda"
                }
            }
        }
    ]
}
```

This policy denies all S3 actions unless they come from a specific Lambda function with ARN `arn:aws:lambda:*:123456789012:function:source_lambda`, or unless they come from the specified VPC. The `StringNotEqualsIfExists` operator tells IAM to process this condition only if the `aws:SourceVpc` key is present in the request. Similarly, IAM considers the `ArnNotEqualsIfExists` operator only if the `lambda:SourceFunctionArn` exists.
Identity-based IAM policies for Lambda

You can use identity-based policies in AWS Identity and Access Management (IAM) to grant users in your account access to Lambda. Identity-based policies can apply to users directly, or to groups and roles that are associated with a user. You can also grant users in another account permission to assume a role in your account and access your Lambda resources. This page shows an example of how identity-based policies can be used for function development.

Lambda provides AWS managed policies that grant access to Lambda API actions and, in some cases, access to other AWS services used to develop and manage Lambda resources. Lambda updates these managed policies as needed to ensure that your users have access to new features when they're released.

- **AWSLambda_FullAccess** – Grants full access to Lambda actions and other AWS services used to develop and maintain Lambda resources. This policy was created by scoping down the previous policy **AWSLambdaFullAccess**.
- **AWSLambda_ReadOnlyAccess** – Grants read-only access to Lambda resources. This policy was created by scoping down the previous policy **AWSLambdaReadOnlyAccess**.
- **AWSLambdaRole** – Grants permissions to invoke Lambda functions.

AWS managed policies grant permission to API actions without restricting the Lambda functions or layers that a user can modify. For finer-grained control, you can create your own policies that limit the scope of a user's permissions.

**Sections**

- Function development
- Layer development and use
- Cross-account roles
- Condition keys for VPC settings

**Function development**

Use identity-based policies to allow users to perform operations on Lambda functions.
Note

For a function defined as a container image, the user permission to access the image MUST be configured in the Amazon Elastic Container Registry. For an example, see Amazon ECR permissions.

The following shows an example of a permissions policy with limited scope. It allows a user to create and manage Lambda functions named with a designated prefix (`intern-`), and configured with a designated execution role.

Example Function development policy

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ReadOnlyPermissions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetAccountSettings",
                "lambda:GetEventSourceMapping",
                "lambda:GetFunction",
                "lambda:GetFunctionConfiguration",
                "lambda:GetFunctionCodeSigningConfig",
                "lambda:GetFunctionConcurrency",
                "lambda:ListEventSourceMappings",
                "lambda:ListFunctions",
                "lambda:ListTags",
                "iam:ListRoles"
            ],
            "Resource": "*"
        },
        {
            "Sid": "DevelopFunctions",
            "Effect": "Allow",
            "NotAction": [
                "lambda:AddPermission",
                "lambda:PutFunctionConcurrency"
            ],
            "Resource": "arn:aws:lambda:*:*:function:intern-*"
        }
    ]
}
```
The permissions in the policy are organized into statements based on the resources and conditions that they support.
• **ReadOnlyPermissions** – The Lambda console uses these permissions when you browse and view functions. They don't support resource patterns or conditions.

```json
"Action": [
  "lambda:GetAccountSettings",
  "lambda:GetEventSourceMapping",
  "lambda:GetFunction",
  "lambda:GetFunctionConfiguration",
  "lambda:GetFunctionCodeSigningConfig",
  "lambda:GetFunctionConcurrency",
  "lambda:ListEventSourceMappings",
  "lambda:ListFunctions",
  "lambda:ListTags",
  "iam:ListRoles"
],
"Resource": "*"
```

• **DevelopFunctions** – Use any Lambda action that operates on functions prefixed with `intern-`, *except* AddPermission and PutFunctionConcurrency. AddPermission modifies the [resource-based policy](https://docs.aws.amazon.com/lambda/latest/dg/access-control-resource-based-policy.html) on the function and can have security implications. PutFunctionConcurrency reserves scaling capacity for a function and can take capacity away from other functions.

```json
"NotAction": [
  "lambda:AddPermission",
  "lambda:PutFunctionConcurrency"
],
"Resource": "arn:aws:lambda::*:*:function:intern-*"
```

• **DevelopEventSourceMappings** – Manage event source mappings on functions that are prefixed with `intern-`. These actions operate on event source mappings, but you can restrict them by function with a *condition*.

```json
"Action": [
  "lambda:DeleteEventSourceMapping",
  "lambda:UpdateEventSourceMapping",
  "lambda:CreateEventSourceMapping"
],
"Resource": "*",
```

**Function development**
• **PassExecutionRole** – View and pass only a role named `intern-lambda-execution-role`, which must be created and managed by a user with IAM permissions. **PassRole** is used when you assign an execution role to a function.

```
"Action": [
    "iam:ListRolePolicies",
    "iam:ListAttachedRolePolicies",
    "iam:GetRole",
    "iam:GetRolePolicy",
    "iam:PassRole",
    "iam:SimulatePrincipalPolicy"
],
"Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
```

• **ViewLogs** – Use CloudWatch Logs to view logs for functions that are prefixed with `intern-`.

```
"Action": [
    "logs:*"
],
"Resource": "arn:aws:logs::*:*:log-group:/aws/lambda/intern-*"
```

This policy allows a user to get started with Lambda, without putting other users' resources at risk. It doesn't allow a user to configure a function to be triggered by or call other AWS services, which requires broader IAM permissions. It also doesn't include permission to services that don't support limited-scope policies, like CloudWatch and X-Ray. Use the read-only policies for these services to give the user access to metrics and trace data.

When you configure triggers for your function, you need access to use the AWS service that invokes your function. For example, to configure an Amazon S3 trigger, you need permission to use the Amazon S3 actions that manage bucket notifications. Many of these permissions are included in the **AWSLambdaFullAccess** managed policy. Example policies are available in this guide's [GitHub repository](https://github.com/awsdocs/aws-lambda-developer-guide).
Layer development and use

The following policy grants a user permission to create layers and use them with functions. The resource patterns allow the user to work in any AWS Region and with any layer version, as long as the name of the layer starts with test-.

Example layer development policy

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "PublishLayers",
         "Effect": "Allow",
         "Action": [
            "lambda:PublishLayerVersion"
         ],
         "Resource": "arn:aws:lambda:*:*:layer:test-*"
      },
      {
         "Sid": "ManageLayerVersions",
         "Effect": "Allow",
         "Action": [
            "lambda:GetLayerVersion",
            "lambda:DeleteLayerVersion"
         ],
         "Resource": "arn:aws:lambda:*:*:layer:test-*:*
      }
   ]
}
```

You can also enforce layer use during function creation and configuration with the `lambda:Layer` condition. For example, you can prevent users from using layers published by other accounts. The following policy adds a condition to the `CreateFunction` and `UpdateFunctionConfiguration` actions to require that any layers specified come from account 123456789012.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "ConfigureFunctions",
         "Effect": "Allow",
         "Action": [
            "lambda:GetLayerVersion",
            "lambda:DeleteLayerVersion"
         ],
         "Resource": "arn:aws:lambda:*:*:layer:test-*:*
      }
   ]
}
```
To ensure that the condition applies, verify that no other statements grant the user permission to these actions.

**Cross-account roles**

You can apply any of the preceding policies and statements to a role, which you can then share with another account to give it access to your Lambda resources. Unlike a user, a role doesn't have credentials for authentication. Instead, it has a trust policy that specifies who can assume the role and use its permissions.

You can use cross-account roles to give accounts that you trust access to Lambda actions and resources. If you just want to grant permission to invoke a function or use a layer, use resource-based policies instead.


**Condition keys for VPC settings**

You can use condition keys for VPC settings to provide additional permission controls for your Lambda functions. For example, you can enforce that all Lambda functions in your organization are connected to a VPC. You can also specify the subnets and security groups that the functions are allowed to use, or are denied from using.

For more information, see [the section called “Using IAM condition keys for VPC settings”](https://docs.aws.amazon.com/IAM/latest/UserGuide/IAM policies examples.html).
Attribute-based access control for Lambda

With attribute-based access control (ABAC), you can use tags to control access to your Lambda functions. You can attach tags to a Lambda function, pass them in certain API requests, or attach them to the AWS Identity and Access Management (IAM) principal making the request. For more information about how AWS grants attribute-based access, see Controlling access to AWS resources using tags in the IAM User Guide.

You can use ABAC to grant least privilege without specifying an Amazon Resource Name (ARN) or ARN pattern in the IAM policy. Instead, you can specify a tag in the condition element of an IAM policy to control access. Scaling is easier with ABAC because you don’t have to update your IAM policies when you create new functions. Instead, add tags to the new functions to control access.

In Lambda, tags work at the function level. Tags aren't supported for layers, code signing configurations, or event source mappings. When you tag a function, those tags apply to all versions and aliases associated with the function. For information about how to tag functions, see Using tags on Lambda functions.

You can use the following condition keys to control function actions:

- **aws:ResourceTag/tag-key**: Control access based on the tags that are attached to Lambda functions.
- **aws:RequestTag/tag-key**: Require tags to be present in a request, such as when creating a new function.
- **aws:PrincipalTag/tag-key**: Control what the IAM principal (the person making the request) is allowed to do based on the tags that are attached to their IAM user or role.
- **aws:TagKeys**: Control whether specific tag keys can be used in a request.

For a complete list of Lambda actions that support ABAC, see Function actions and check the Condition column in the table.

The following steps demonstrate one way to set up permissions using ABAC. In this example scenario, you'll create four IAM permissions policies. Then, you'll attach these policies to a new IAM role. Finally, you'll create an IAM user and give that user permission to assume the new role.
Prerequisites

Make sure that you have a Lambda execution role. You'll use this role when you grant IAM permissions and when you create a Lambda function.

Step 1: Require tags on new functions

When using ABAC with Lambda, it's a best practice to require that all functions have tags. This helps ensure that your ABAC permissions policies work as expected.

Create an IAM policy similar to the following example. This policy uses the aws:RequestTag/tag-key, aws:ResourceTag/tag-key, and aws:TagKeys condition keys to require that new functions and the IAM principal creating the functions both have the project tag. The ForAllValues modifier ensures that project is the only allowed tag. If you don't include the ForAllValues modifier, users can add other tags to the function as long as they also pass project.

Example – Require tags on new functions

```json
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": [
      "lambda:CreateFunction",
      "lambda:TagResource"
    ],
    "Resource": "arn:aws:lambda::*:*:function:*",
    "Condition": {
      "StringEquals": {
        "aws:RequestTag/project": "${aws:PrincipalTag/project}",
        "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
      },
      "ForAllValues:StringEquals": {
        "aws:TagKeys": "project"
      }
    }
  }
}
```
Step 2: Allow actions based on tags attached to a Lambda function and IAM principal

Create a second IAM policy using the `aws:ResourceTag/tag-key` condition key to require the principal's tag to match the tag that's attached to the function. The following example policy allows principals with the `project` tag to invoke functions with the `project` tag. If a function has any other tags, the action is denied.

Example – Require matching tags on function and IAM principal

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "lambda:InvokeFunction",
            "lambda:GetFunction"
         ],
         "Resource": "arn:aws:lambda:*:*:function:*",
         "Condition": {
            "StringEquals": {
               "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
            }
         }
      }
   ]
}
```

Step 3: Grant list permissions

Create a policy that allows the principal to list Lambda functions and IAM roles. This allows the principal to see all Lambda functions and IAM roles on the console and when calling the API actions.

Example – Grant Lambda and IAM list permissions

```json
{
   "Version": "2012-10-17",
   "Statement": [
```
Step 4: Grant IAM permissions

Create a policy that allows `iam:PassRole`. This permission is required when you assign an execution role to a function. In the following example policy, replace the example ARN with the ARN of your Lambda execution role.

Note

Do not use the `ResourceTag` condition key in a policy with the `iam:PassRole` action. You cannot use the tag on an IAM role to control access to who can pass that role. For more information about permissions required to pass a role to a service, see [Granting a user permissions to pass a role to an AWS service](#).

Example – Grant permission to pass the execution role

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": ["iam:PassRole"],
      "Resource": "arn:aws:iam::111122223333:role/lambda-ex"
    }
  ]
}
```
Step 5: Create the IAM role

It's a best practice to use roles to delegate permissions. Create an IAM role called abac-project-role:

- On Step 1: Select trusted entity: Choose AWS account and then choose This account.
- On Step 2: Add permissions: Attach the four IAM policies that you created in the previous steps.
- On Step 3: Name, review, and create: Choose Add tag. For Key, enter project. Don't enter a Value.

Step 6: Create the IAM user

Create an IAM user called abac-test-user. In the Set permissions section, choose Attach existing policies directly and then choose Create policy. Enter the following policy definition. Replace 111122223333 with your AWS account ID. This policy allows abac-test-user to assume abac-project-role.

Example – Allow IAM user to assume ABAC role

```json
{
   "Version": "2012-10-17",
   "Statement": {
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Resource": "arn:aws:iam::111122223333:role/abac-project-role"
   }
}
```

Step 7: Test the permissions

1. Sign in to the AWS console as abac-test-user. For more information, see Sign in as an IAM user.
2. Switch to the abac-project-role role. For more information, see Switching to a role (console).
3. Create a Lambda function:
• Under Permissions, choose Change default execution role, and then for Execution role, choose Use an existing role. Choose the same execution role that you used in Step 4: Grant IAM permissions.

• Under Advanced settings, choose Enable tags and then choose Add new tag. For Key, enter project. Don't enter a Value.

4. Test the function.

5. Create a second Lambda function and add a different tag, such as environment. This operation should fail because the ABAC policy that you created in Step 1: Require tags on new functions only allows the principal to create functions with the project tag.

6. Create a third function without tags. This operation should fail because the ABAC policy that you created in Step 1: Require tags on new functions doesn't allow the principal to create functions without tags.

This authorization strategy allows you to control access without creating new policies for each new user. To grant access to new users, simply give them permission to assume the role that corresponds to their assigned project.
Using resource-based policies for Lambda

Lambda supports resource-based permissions policies for Lambda functions and layers. Resource-based policies let you grant usage permission to other AWS accounts or organizations on a per-resource basis. You also use a resource-based policy to allow an AWS service to invoke your function on your behalf.

For Lambda functions, you can grant an account permission to invoke or manage a function. You can also use a single resource-based policy to grant permissions to an entire organization in AWS Organizations. You can also use resource-based policies to grant invoke permission to an AWS service that invokes a function in response to activity in your account.

To view a function's resource-based policy

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose Configuration and then choose Permissions.
4. Scroll down to Resource-based policy and then choose View policy document. The resource-based policy shows the permissions that are applied when another account or AWS service attempts to access the function. The following example shows a statement that allows Amazon S3 to invoke a function named my-function for a bucket named my-bucket in account 123456789012.

Example Resource-based policy

```json
{
  "Version": "2012-10-17",
  "Id": "default",
  "Statement": [
    {
      "Sid": "lambda-allow-s3-my-function",
      "Effect": "Allow",
      "Principal": {
        "Service": "s3.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Condition": {
        "StringEquals": {
```
For Lambda layers, you can only use a resource-based policy on a specific layer version, instead of the entire layer. In addition to policies that grant permission to a single account or multiple accounts, for layers, you can also grant permission to all accounts in an organization.

**Note**
You can only update resource-based policies for Lambda resources within the scope of the `AddPermission` and `AddLayerVersionPermission` API actions. Currently, you can't author policies for your Lambda resources in JSON, or use conditions that don't map to parameters for those actions.

Resource-based policies apply to a single function, version, alias, or layer version. They grant permission to one or more services and accounts. For trusted accounts that you want to have access to multiple resources, or to use API actions that resource-based policies don't support, you can use cross-account roles.

Topics

- Supported API actions
- Granting function access to AWS services
- Granting function access to an organization
- Granting function access to other accounts
- Granting layer access to other accounts
- Cleaning up resource-based policies
Supported API actions

The following Lambda API actions support resource-based policies:

- CreateAlias
- DeleteAlias
- DeleteFunction
- DeleteFunctionConcurrency
- DeleteFunctionEventInvokeConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias
- GetFunction
- GetFunctionConcurrency
- GetFunctionConfiguration
- GetFunctionEventInvokeConfig
- GetPolicy
- GetProvisionedConcurrencyConfig
- Invoke
- ListAliases
- ListFunctionEventInvokeConfigs
- ListProvisionedConcurrencyConfigs
- ListTags
- ListVersionsByFunction
- PublishVersion
- PutFunctionConcurrency
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig
- TagResource
- UntagResource
- UpdateAlias
- UpdateFunctionCode
Granting function access to AWS services

When you use an AWS service to invoke your function, you grant permission in a statement on a resource-based policy. You can apply the statement to the entire function to be invoked or managed, or limit the statement to a single version or alias.

Note

When you add a trigger to your function with the Lambda console, the console updates the function's resource-based policy to allow the service to invoke it. To grant permissions to other accounts or services that aren't available in the Lambda console, you can use the AWS CLI.

Add a statement with the add-permission command. The simplest resource-based policy statement allows a service to invoke a function. The following command grants Amazon SNS permission to invoke a function named my-function.

```
aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --statement-id sns --principal sns.amazonaws.com --output text
```

You should see the following output:

```
{"Sid":"sns","Effect":"Allow","Principal":
```

This lets Amazon SNS call the lambda:Invoke API for the function, but it doesn't restrict the Amazon SNS topic that triggers the invocation. To ensure that your function is only invoked by a specific resource, specify the Amazon Resource Name (ARN) of the resource with the source-arn option. The following command only allows Amazon SNS to invoke the function for subscriptions to a topic named my-topic.

```
aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --statement-id sns-my-topic
```
Some services can invoke functions in other accounts. If you specify a source ARN that has your account ID in it, that isn't an issue. For Amazon S3, however, the source is a bucket whose ARN doesn't have an account ID in it. It's possible that you could delete the bucket and another account could create a bucket with the same name. Use the source-account option with your account ID to ensure that only resources in your account can invoke the function.

```bash
aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --statement-id s3-account \
--principal s3.amazonaws.com --source-arn arn:aws:s3:::my-bucket-123456 --source-account 123456789012
```

### Granting function access to an organization

To grant permissions to an organization in AWS Organizations, specify the organization ID as the principal-org-id. The following AddPermission AWS CLI command grants invocation access to all users in organization o-a1b2c3d4e5f.

```bash
aws lambda add-permission --function-name example \
--statement-id PrincipalOrgIDEexample --action lambda:InvokeFunction \
--principal * --principal-org-id o-a1b2c3d4e5f
```

#### Note

In this command, Principal is *. This means that all users in the organization o-a1b2c3d4e5f get function invocation permissions. If you specify an AWS account or role as the Principal, then only that principal gets function invocation permissions, but only if they are also part of the o-a1b2c3d4e5f organization.

This command creates a resource-based policy that looks like the following:

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "PrincipalOrgIDEexample",
         "Effect": "Allow",
```
"Principal": "*",
"Action": "lambda:InvokeFunction",
"Condition": {
   "StringEquals": {
      "aws:PrincipalOrgID": "o-a1b2c3d4e5f"
   }
}
]


### Granting function access to other accounts

To grant permissions to another AWS account, specify the account ID as the principal. The following example grants account 111122223333 permission to invoke my-function with the prod alias.

```bash
aws lambda add-permission --function-name my-function:prod --statement-id xaccount --action lambda:InvokeFunction --principal 111122223333 --output text
```

You should see the following output:

```json
{"Sid":"xaccount","Effect":"Allow","Principal":
```

The resource-based policy grants permission for the other account to access the function, but doesn't allow users in that account to exceed their permissions. Users in the other account must have the corresponding [user permissions](https://docs.aws.amazon.com/IdentityAndAccessManagement/latest/UserGuide/permissions.html) to use the Lambda API.

To limit access to a user or role in another account, specify the full ARN of the identity as the principal. For example, `arn:aws:iam::123456789012:user/developer`.

The alias limits which version the other account can invoke. It requires the other account to include the alias in the function ARN.
You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "1"
}
```

The function owner can then update the alias to point to a new version without the caller needing to change the way they invoke your function. This ensures that the other account doesn't need to change its code to use the new version, and it only has permission to invoke the version of the function associated with the alias.

You can grant cross-account access for most API actions that operate on an existing function. For example, you could grant access to `lambda:ListAliases` to let an account get a list of aliases, or `lambda:GetFunction` to let them download your function code. Add each permission separately, or use `lambda:*` to grant access to all actions for the specified function.

To grant other accounts permission for multiple functions, or for actions that don't operate on a function, we recommend that you use IAM roles.

### Granting layer access to other accounts

To grant layer-usage permission to another account, add a statement to the layer version's permissions policy using the `add-layer-version-permission` command. In each statement, you can grant permission to a single account, all accounts, or an organization.

The following example grants account 111122223333 access to version 2 of the bash-runtime layer.

```
aws lambda add-layer-version-permission --layer-name bash-runtime --statement-id xaccount --action lambda:GetLayerVersion --principal 111122223333 --version-number 2 --output text
```

You should see output similar to the following:
AWS Lambda

Developer Guide

e210ffdc-e901-43b0-824b-5fcd0dd26d16
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}

Permissions apply only to a single layer version. Repeat the process each time that you create a
new layer version.
To grant permission to all accounts in an organization, use the organization-id option. The
following example grants all accounts in an organization permission to use version 3 of a layer.
aws lambda add-layer-version-permission --layer-name my-layer \
--statement-id engineering-org --version-number 3 --principal '*' \
--action lambda:GetLayerVersion --organization-id o-t194hfs8cz --output text

You should see the following output:

b0cd9796-d4eb-4564-939f-de7fe0b42236
{"Sid":"engineeringorg","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:lam
east-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"

To grant permission to all AWS accounts, use * for the principal, and omit the organization ID. For
multiple accounts or organizations, you need to add multiple statements.

Cleaning up resource-based policies
To view a function's resource-based policy, use the get-policy command.
aws lambda get-policy --function-name my-function --output text

You should see the following output:
{"Version":"2012-10-17","Id":"default","Statement":
[{"Sid":"sns","Effect":"Allow","Principal":
7c681fc9b791-4e91-acdf-eb847fdaa0f0

For versions and aliases, append the version number or alias to the function name.
Cleaning up resource-based policies

1430


AWS Lambda

Developer Guide

aws lambda get-policy --function-name my-function:PROD

To remove permissions from your function, use remove-permission.
aws lambda remove-permission --function-name example --statement-id sns

Use the get-layer-version-policy command to view the permissions on a layer.
aws lambda get-layer-version-policy --layer-name my-layer --version-number 3 --output
text

You should see the following output:

b0cd9796-d4eb-4564-939f-de7fe0b42236
{"Sid":"engineeringorg","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:lam
west-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"

Use remove-layer-version-permission to remove statements from the policy.
aws lambda remove-layer-version-permission --layer-name my-layer --version-number 3 -statement-id engineering-org

Cleaning up resource-based policies

1431


Resources and conditions for Lambda actions

You can restrict the scope of a user's permissions by specifying resources and conditions in an AWS Identity and Access Management (IAM) policy. Each action in a policy supports a combination of resource and condition types that varies depending on the behavior of the action.

Every IAM policy statement grants permission to an action that's performed on a resource. When the action doesn't act on a named resource, or when you grant permission to perform the action on all resources, the value of the resource in the policy is a wildcard (*). For many actions, you can restrict the resources that a user can modify by specifying the Amazon Resource Name (ARN) of a resource, or an ARN pattern that matches multiple resources.

To restrict permissions by resource, specify the resource by ARN.

Lambda resource ARN format

- Event source mapping – arn:aws:lambda:us-west-2:123456789012:event-source-mapping:fa123456-14a1-4fd2-9fec-83de64ad683de6d47

For example, the following policy allows a user in AWS account 123456789012 to invoke a function named my-function in the US West (Oregon) AWS Region.

Example invoke function policy

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Invoke",
            "Effect": "Allow",
            "Action": [
```
This is a special case where the action identifier (lambda:InvokeFunction) differs from the API operation (Invoke). For other actions, the action identifier is the operation name prefixed by lambda:.

Sections

- Policy conditions
- Function resource names
- Function actions
- Event source mapping actions
- Layer actions

Policy conditions

Conditions are an optional policy element that applies additional logic to determine if an action is allowed. In addition to common conditions that all actions support, Lambda defines condition types that you can use to restrict the values of additional parameters on some actions.

For example, the lambda:Principal condition lets you restrict the service or account that a user can grant invocation access to on a function's resource-based policy. The following policy lets a user grant permission to Amazon Simple Notification Service (Amazon SNS) topics to invoke a function named test.

Example manage function policy permissions

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "ManageFunctionPolicy",
         "Effect": "Allow",
         "Action": [
            "lambda:AddPermission",
            "lambda:InvokeFunction"
         ],
      }
   ]
}
```
The condition requires that the principal is Amazon SNS and not another service or account. The resource pattern requires that the function name is test and includes a version number or alias. For example, test:v1.

For more information on resources and conditions for Lambda and other AWS services, see [Actions, resources, and condition keys for AWS services](https://docs.aws.amazon.com/IAM/latest/UserGuide/auth-resource-condition-keys.html) in the *Service Authorization Reference*.

**Function resource names**

You reference a Lambda function in a policy statement using an Amazon Resource Name (ARN). The format of a function ARN depends on whether you are referencing the whole function (unqualified) or a function version or alias (qualified).

When making Lambda API calls, users can specify a version or alias by passing a version ARN or alias ARN in the `GetFunction` FunctionName parameter, or by setting a value in the `GetFunction` Qualifier parameter. Lambda makes authorization decisions by comparing the resource element in the IAM policy with both the FunctionName and Qualifier passed in API calls. If there is a mismatch, Lambda denies the request.

Whether you are allowing or denying an action on your function, you must use the correct function ARN types in your policy statement to achieve the results that you expect. For example, if your policy references the unqualified ARN, Lambda accepts requests that reference the unqualified ARN but denies requests that reference a qualified ARN.

**Note**

You can't use a wildcard character (*) to match the account ID. For more information on accepted syntax, see [IAM JSON policy reference](https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_json_examples.html) in the *IAM User Guide*.
Example allowing invocation of an unqualified ARN

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": "lambda:InvokeFunction",
      }
   ]
}
```

If your policy references a specific qualified ARN, Lambda accepts requests that reference that ARN but denies requests that reference the unqualified ARN or a different qualified ARN, for example, `myFunction:2`.

Example allowing invocation of a specific qualified ARN

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": "lambda:InvokeFunction",
      }
   ]
}
```

If your policy references any qualified ARN using `:*`, Lambda accepts any qualified ARN but denies requests that reference the unqualified ARN.

Example allowing invocation of any qualified ARN

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": "lambda:InvokeFunction",
      }
   ]
}
```
If your policy references any ARN using *, Lambda accepts any qualified or unqualified ARN.

**Example allowing invocation of any qualified or unqualified ARN**

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
        }
    ]
}
```

**Function actions**

Actions that operate on a function can be restricted to a specific function by function, version, or alias ARN, as described in the following table. Actions that don't support resource restrictions are granted for all resources (*).

### Function actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Resource</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddPermission</td>
<td>Function</td>
<td>lambda:Principal</td>
</tr>
<tr>
<td>RemovePermission</td>
<td>Function version</td>
<td>aws:ResourceTag/${TagKey}</td>
</tr>
<tr>
<td></td>
<td>Function alias</td>
<td>lambda:FunctionUrl AuthType</td>
</tr>
<tr>
<td>Invoke</td>
<td>Function</td>
<td>aws:ResourceTag/${TagKey}</td>
</tr>
<tr>
<td>Permission: lambda:InvokeFunction</td>
<td>Function version</td>
<td>lambda:EventSourceToken</td>
</tr>
<tr>
<td></td>
<td>Function alias</td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>Resource</td>
<td>Condition</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>CreateFunction</strong></td>
<td>Function</td>
<td>lambda:CodeSigning ConfigArn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lambda:Layer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lambda:VpcIds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lambda:SubnetIds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lambda:SecurityGroupIds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aws:ResourceTag/${TagKey}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aws:RequestTag/${TagKey}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aws:TagKeys</td>
</tr>
<tr>
<td><strong>UpdateFunctionConfiguration</strong></td>
<td>Function</td>
<td>lambda:CodeSigning ConfigArn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lambda:Layer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lambda:VpcIds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lambda:SubnetIds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lambda:SecurityGroupIds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aws:ResourceTag/${TagKey}</td>
</tr>
<tr>
<td>Action</td>
<td>Resource</td>
<td>Condition</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>--------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>CreateAlias</td>
<td>Function</td>
<td>aws:ResourceTag/${TagKey}</td>
</tr>
<tr>
<td>DeleteAlias</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeleteFunction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeleteFunctionCodeSigningConfig</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeleteFunctionConcurrency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetAlias</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetFunction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetFunctionCodeSigningConfig</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetFunctionConcurrency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetFunctionConfiguration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetPolicy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ListProvisionedConcurrencyConfigs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ListAliases</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ListTags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ListVersionsByFunction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PublishVersion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PutFunctionCodeSigningConfig</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PutFunctionConcurrency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateAlias</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateFunctionCode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>Resource</td>
<td>Condition</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>---------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td><strong>CreateFunctionUrlConfig</strong></td>
<td>Function</td>
<td>lambda:FunctionUrl AuthType</td>
</tr>
<tr>
<td></td>
<td>Function alias</td>
<td>lambda:FunctionArn</td>
</tr>
<tr>
<td><strong>DeleteFunctionUrlConfig</strong></td>
<td></td>
<td>aws:ResourceTag/${TagKey}</td>
</tr>
<tr>
<td><strong>GetFunctionUrlConfig</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UpdateFunctionUrlConfig</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ListFunctionUrlConfigs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DeleteFunctionEventInvokeConfig</strong></td>
<td>Function</td>
<td>aws:ResourceTag/${TagKey}</td>
</tr>
<tr>
<td><strong>GetFunctionEventInvokeConfig</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ListFunctionEventInvokeConfigs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PutFunctionEventInvokeConfig</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UpdateFunctionEventInvokeConfig</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DeleteProvisionedConcurrencyConfig</strong></td>
<td>Function alias</td>
<td>aws:ResourceTag/${TagKey}</td>
</tr>
<tr>
<td><strong>GetProvisionedConcurrencyConfig</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PutProvisionedConcurrencyConfig</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>GetAccountSettings</strong></td>
<td>*</td>
<td>None</td>
</tr>
<tr>
<td><strong>ListFunctions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TagResource</strong></td>
<td>Function</td>
<td>aws:ResourceTag/${TagKey}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aws:RequestTag/${TagKey}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aws:TagKeys</td>
</tr>
</tbody>
</table>
### Event source mapping actions

For **event source mappings**, you can restrict delete and update permissions to a specific event source. The `lambda:FunctionArn` condition lets you restrict which functions a user can configure an event source to invoke.

For these actions, the resource is the event source mapping, so Lambda provides a condition that lets you restrict permission based on the function that the event source mapping invokes.

<table>
<thead>
<tr>
<th>Action</th>
<th>Resource</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DeleteEventSourceMapping</strong></td>
<td>Event source mapping</td>
<td><code>lambda:FunctionArn</code></td>
</tr>
<tr>
<td><strong>UpdateEventSourceMapping</strong></td>
<td>Event source mapping</td>
<td><code>lambda:FunctionArn</code></td>
</tr>
<tr>
<td><strong>CreateEventSourceMapping</strong></td>
<td>*</td>
<td><code>lambda:FunctionArn</code></td>
</tr>
<tr>
<td><strong>GetEventSourceMapping</strong></td>
<td>*</td>
<td>None</td>
</tr>
<tr>
<td><strong>ListEventSourceMappings</strong></td>
<td>*</td>
<td>None</td>
</tr>
</tbody>
</table>

### Layer actions

Layer actions let you restrict the layers that a user can manage or use with a function. Actions related to layer use and permissions act on a version of a layer, while `PublishLayerVersion` acts on a layer name. You can use either with wildcards to restrict the layers that a user can work with by name.
Note

The GetLayerVersion action also covers GetLayerVersionByArn. Lambda does not support GetLayerVersionByArn as an IAM action.

Layer actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Resource</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddLayerVersionPermission</td>
<td>Layer version</td>
<td>None</td>
</tr>
<tr>
<td>RemoveLayerVersionPermission</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetLayerVersion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetLayerVersionPolicy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeleteLayerVersion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ListLayerVersions</td>
<td>Layer</td>
<td>None</td>
</tr>
<tr>
<td>PublishLayerVersion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ListLayers</td>
<td>*</td>
<td>None</td>
</tr>
</tbody>
</table>
Using permissions boundaries for AWS Lambda applications

When you create an application in the AWS Lambda console, Lambda applies a permissions boundary to the application's IAM roles. The permissions boundary limits the scope of the execution role that the application's template creates for each of its functions, and any roles that you add to the template. The permissions boundary prevents users with write access to the application's Git repository from escalating the application's permissions beyond the scope of its own resources.

The application templates in the Lambda console include a global property that applies a permissions boundary to all functions that they create.

```
Globals:
  Function:
    PermissionsBoundary: !Sub 'arn:${AWS::Partition}:iam::${AWS::AccountId}:policy/${AppId}-${AWS::Region}-PermissionsBoundary'
```

The boundary limits the permissions of the functions' roles. You can add permissions to a function's execution role in the template, but that permission is only effective if it's also allowed by the permissions boundary. The role that AWS CloudFormation assumes to deploy the application enforces the use of the permissions boundary. That role only has permission to create and pass roles that have the application's permissions boundary attached.

By default, an application's permissions boundary enables functions to perform actions on the resources in the application. For example, if the application includes an Amazon DynamoDB table, the boundary allows access to any API action that can be restricted to operate on specific tables with resource-level permissions. You can only use actions that don't support resource-level permissions if they're specifically permitted in the boundary. These include Amazon CloudWatch Logs and AWS X-Ray API actions for logging and tracing.

Example permissions boundary

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "*"
      ],
      "Resource": [
```
To access other resources or API actions, you or an administrator must expand the permissions boundary to include those resources. You might also need to update the execution role or deployment role of an application to allow the use of additional actions.

- **Permissions boundary** – Extend the application's permissions boundary when you add resources to your application, or the execution role needs access to more actions. In IAM, add resources to the boundary to allow the use of API actions that support resource-level permissions on that resource's type. For actions that don't support resource-level permissions, add them in a statement that isn't scoped to any resource.

- **Execution role** – Extend a function's execution role when it needs to use additional actions. In the application template, add policies to the execution role. The intersection of permissions in the boundary and execution role is granted to the function.

- **Deployment role** – Extend the application's deployment role when it needs additional permissions to create or configure resources. In IAM, add policies to the application's deployment
role. The deployment role needs the same user permissions that you need to deploy or update an application in AWS CloudFormation.

For a tutorial that walks through adding resources to an application and extending its permissions, see ???.

For more information, see Permissions boundaries for IAM entities in the IAM User Guide.
Security in AWS Lambda

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security of the cloud and security in the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS compliance programs. To learn about the compliance programs that apply to AWS Lambda, see AWS Services in Scope by Compliance Program.

- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company’s requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Lambda. The following topics show you how to configure Lambda to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Lambda resources.

For more information about applying security principles to Lambda applications, see Security in Serverless Land.

Topics

- [Data protection in AWS Lambda](#)
- [Identity and access management for Lambda](#)
- [Governance for AWS Lambda](#)
- [Compliance validation for AWS Lambda](#)
- [Resilience in AWS Lambda](#)
- [Infrastructure security in AWS Lambda](#)
Data protection in AWS Lambda

The AWS shared responsibility model applies to data protection in AWS Lambda. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the AWS Security Blog.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-2.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a Name field. This includes when you work with Lambda or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Sections

- Encryption in transit
- Encryption at rest
Encryption in transit

Lambda API endpoints only support secure connections over HTTPS. When you manage Lambda resources with the AWS Management Console, AWS SDK, or the Lambda API, all communication is encrypted with Transport Layer Security (TLS). For a full list of API endpoints, see [AWS Regions and endpoints](https://docs.aws.amazon.com/lambda/latest/dg/lambda-api-endpoints.html) in the AWS General Reference.

When you **connect your function to a file system**, Lambda uses encryption in transit for all connections. For more information, see [Data encryption in Amazon EFS](https://docs.aws.amazon.com/efs/latest/ug/encrypt-data.html) in the *Amazon Elastic File System User Guide*.

When you use **environment variables**, you can enable console encryption helpers to use client-side encryption to protect the environment variables in transit. For more information, see [Securing environment variables](https://docs.aws.amazon.com/lambda/latest/dg/configuration-envvars.html).

Encryption at rest

You can use **environment variables** to store secrets securely for use with Lambda functions. Lambda always encrypts environment variables at rest. By default, Lambda uses an AWS KMS key that Lambda creates in your account to encrypt your environment variables. This AWS managed key is named `aws/lambda`.

On a per-function basis, you can optionally configure Lambda to use a customer managed key instead of the default AWS managed key to encrypt your environment variables. For more information, see [Securing environment variables](https://docs.aws.amazon.com/lambda/latest/dg/configuration-envvars.html).

Lambda always encrypts files that you upload to Lambda, including **deployment packages** and **layer archives**.

Amazon CloudWatch Logs and AWS X-Ray also encrypt data by default, and can be configured to use a customer managed key. For details, see [Encrypt log data in CloudWatch Logs](https://docs.aws.amazon.com/lambda/latest/dg/lambda-log-streams.html) and [Data protection in AWS X-Ray](https://docs.aws.amazon.com/xray/latest/userguide/xray-monitoring-console.html).

**Identity and access management for Lambda**

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be **authenticated** (signed in) and **authorized** (have permissions) to use Lambda resources. IAM is an AWS service that you can use with no additional charge.
Topics

• Audience

• Authenticating with identities

• Managing access using policies

• How AWS Lambda works with IAM

• AWS Lambda identity-based policy examples

• AWS managed policies for AWS Lambda

• Troubleshooting AWS Lambda identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in Lambda.

Service user – If you use the Lambda service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Lambda features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in Lambda, see Troubleshooting AWS Lambda identity and access.

Service administrator – If you’re in charge of Lambda resources at your company, you probably have full access to Lambda. It’s your job to determine which Lambda features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with Lambda, see How AWS Lambda works with IAM.

IAM administrator – If you’re an IAM administrator, you might want to learn details about how you can write policies to manage access to Lambda. To view example Lambda identity-based policies that you can use in IAM, see AWS Lambda identity-based policy examples.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.
You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see [How to sign in to your AWS account](https://docs.aws.amazon.com/iam/latest/userguide/singin-admin.html) in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see [Signing AWS API requests](https://docs.aws.amazon.com/iam/latest/userguide/signing.html) in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see [Multi-factor authentication](https://docs.aws.amazon.com/iam/latest/userguide/mfa-auth.html) in the *AWS IAM Identity Center User Guide* and [Using multi-factor authentication (MFA) in AWS](https://docs.aws.amazon.com/iam/latest/userguide/mfa-auth.html) in the *IAM User Guide*.

**AWS account root user**

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account root user and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see [Tasks that require root user credentials](https://docs.aws.amazon.com/iam/latest/userguide/rootuser.html) in the *IAM User Guide*.

**Users and groups**

An IAM user is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate
access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the IAM User Guide.

An IAM group is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named IAMAdmins and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see When to create an IAM user (instead of a role) in the IAM User Guide.

IAM roles

An IAM role is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by switching roles. You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Using IAM roles in the IAM User Guide.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Creating a role for a third-party Identity Provider in the IAM User Guide. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the AWS IAM Identity Center User Guide.

- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.

- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the IAM User Guide.
• **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.

• **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see [Forward access sessions](#).

• **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Creating a role to delegate permissions to an AWS service](#) in the IAM User Guide.

• **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

• **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see [Using an IAM role to grant permissions to applications running on Amazon EC2 instances](#) in the IAM User Guide.

To learn whether to use IAM roles or IAM users, see [When to create an IAM role (instead of a user)](#) in the IAM User Guide.

### Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their
permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the IAM User Guide.

Administrators can use AWS JSON policies to specify who has access to what. That is, which principal can perform actions on what resources, and under what conditions.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the iam:GetRole action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

**Identity-based policies**

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM policies in the IAM User Guide.

Identity-based policies can be further categorized as inline policies or managed policies. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choosing between managed policies and inline policies in the IAM User Guide.

**Resource-based policies**

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM role trust policies and Amazon S3 bucket policies. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.
Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

**Access control lists (ACLs)**

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see [Access control list (ACL) overview](https://docs.aws.amazon.com/AmazonS3/latest/dev/access-control-lists.html) in the *Amazon Simple Storage Service Developer Guide*.

**Other policy types**

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the Principal field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see [Permissions boundaries for IAM entities](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_aud-permbounds.html) in the *IAM User Guide*.

- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see [How SCPs work](https://docs.aws.amazon.com/organizations/latest/userguide/orgs_getting_started_scps.html) in the *AWS Organizations User Guide*.

- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see [Session policies](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_temp.html) in the *IAM User Guide*.
Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the IAM User Guide.

How AWS Lambda works with IAM

Before you use IAM to manage access to Lambda, you should understand what IAM features are available to use with Lambda. To get a high-level view of how Lambda and other AWS services work with IAM, see AWS services that work with IAM in the IAM User Guide.

For an overview of permissions, policies, and roles as they are used by Lambda, see Lambda resource access permissions.

AWS Lambda identity-based policy examples

By default, users and roles don't have permission to create or modify Lambda resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An administrator must create IAM policies that grant users and roles permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the users or groups that require those permissions.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see Creating policies on the JSON tab in the IAM User Guide.

Topics

- Policy best practices
- Using the Lambda console
- Allow users to view their own permissions

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Lambda resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- Get started with AWS managed policies and move toward least-privilege permissions – To get started granting permissions to your users and workloads, use the AWS managed policies
that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the IAM User Guide.

- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as least-privilege permissions. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the IAM User Guide.

- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see IAM JSON policy elements: Condition in the IAM User Guide.

- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see IAM Access Analyzer policy validation in the IAM User Guide.

- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Configuring MFA-protected API access in the IAM User Guide.

For more information about best practices in IAM, see Security best practices in IAM in the IAM User Guide.

**Using the Lambda console**

To access the AWS Lambda console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Lambda resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won’t function as intended for entities (users or roles) with that policy.

For an example policy that grants minimal access for function development, see Function development. In addition to Lambda APIs, the Lambda console uses other services to display
trigger configuration and let you add new triggers. If your users use Lambda with other services, they need access to those services as well. For details on configuring other services with Lambda, see Using AWS Lambda with other services.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "ViewOwnUserInfo",
         "Effect": "Allow",
         "Action": [
            "iam:GetUserPolicy",
            "iam:ListGroupsForUser",
            "iam:ListAttachedUserPolicies",
            "iam:ListUserPolicies",
            "iam:GetUser"
         ],
         "Resource": ["arn:aws:iam::*:user/${aws:username}"],
      },
      {
         "Sid": "NavigateInConsole",
         "Effect": "Allow",
         "Action": [
            "iam:GetGroupPolicy",
            "iam:GetPolicyVersion",
            "iam:GetPolicy",
            "iam:ListAttachedGroupPolicies",
            "iam:ListGroupPolicies",
            "iam:ListPolicyVersions",
            "iam:ListPolicies",
            "iam:ListUsers"
         ],
         "Resource": "*"
      }
   ]
}
```
AWS managed policies for AWS Lambda

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining customer managed policies that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see AWS managed policies in the IAM User Guide.

Topics

- AWS managed policy: AWSLambda_FullAccess
- AWS managed policy: AWSLambda_ReadOnlyAccess
- AWS managed policy: AWSLambdaBasicExecutionRole
- AWS managed policy: AWSLambdaDynamoDBExecutionRole
- AWS managed policy: AWSLambdaENIManagementAccess
- AWS managed policy: AWSLambdaExecute
- AWS managed policy: AWSLambdaInvocation-DynamoDB
- AWS managed policy: AWSLambdaKinesisExecutionRole
- AWS managed policy: AWSLambdaMSKExecutionRole
- AWS managed policy: AWSLambdaRole
- AWS managed policy: AWSLambdaSQSQueueExecutionRole
- AWS managed policy: AWSLambdaVPCAccessExecutionRole
- Lambda updates to AWS managed policies
AWS managed policy: AWSLambda_FullAccess

This policy grants full access to Lambda actions. It also grants permissions to other AWS services that are used to develop and maintain Lambda resources.

You can attach the AWSLambda_FullAccess policy to your users, groups, and roles.

Permissions details

This policy includes the following permissions:

- **lambda** – Allows principals full access to Lambda.
- **cloudformation** – Allows principals to describe AWS CloudFormation stacks and list the resources in those stacks.
- **cloudwatch** – Allows principals to list Amazon CloudWatch metrics and get metric data.
- **ec2** – Allows principals to describe security groups, subnets, and VPCs.
- **iam** – Allows principals to get policies, policy versions, roles, role policies, attached role policies, and the list of roles. This policy also allows principals to pass roles to Lambda. The **PassRole** permission is used when you assign an execution role to a function.
- **kms** – Allows principals to list aliases.
- **logs** – Allows principals to describe Amazon CloudWatch log groups. For log groups that are associated with a Lambda function, this policy allows the principal to describe log streams, get log events, and filter log events.
- **states** – Allows principals to describe and list AWS Step Functions state machines.
- **tag** – Allows principals to get resources based on their tags.
- **xray** – Allows principals to get AWS X-Ray trace summaries and retrieve a list of traces specified by ID.

For more information about this policy, including the JSON policy document and policy versions, see [AWSLambda_FullAccess](#) in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AWSLambda_ReadOnlyAccess

This policy grants read-only access to Lambda resources and to other AWS services that are used to develop and maintain Lambda resources.

You can attach the AWSLambda_ReadOnlyAccess policy to your users, groups, and roles.
Permissions details

This policy includes the following permissions:

- **lambda** – Allows principals to get and list all resources.
- **cloudformation** – Allows principals to describe and list AWS CloudFormation stacks and list the resources in those stacks.
- **cloudwatch** – Allows principals to list Amazon CloudWatch metrics and get metric data.
- **ec2** – Allows principals to describe security groups, subnets, and VPCs.
- **iam** – Allows principals to get policies, policy versions, roles, role policies, attached role policies, and the list of roles.
- **kms** – Allows principals to list aliases.
- **logs** – Allows principals to describe Amazon CloudWatch log groups. For log groups that are associated with a Lambda function, this policy allows the principal to describe log streams, get log events, and filter log events.
- **states** – Allows principals to describe and list AWS Step Functions state machines.
- **tag** – Allows principals to get resources based on their tags.
- **xray** – Allows principals to get AWS X-Ray trace summaries and retrieve a list of traces specified by ID.

For more information about this policy, including the JSON policy document and policy versions, see [AWSLambda_ReadOnlyAccess](#) in the *AWS Managed Policy Reference Guide*.

**AWS managed policy: AWSLambdaBasicExecutionRole**

This policy grants permissions to upload logs to CloudWatch Logs.

You can attach the AWSLambdaBasicExecutionRole policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see [AWSLambdaBasicExecutionRole](#) in the *AWS Managed Policy Reference Guide*.

**AWS managed policy: AWSLambdaDynamoDBExecutionRole**

This policy grants permissions to read records from an Amazon DynamoDB stream and write to CloudWatch Logs.
You can attach the AWSLambdaDynamoDBExecutionRole policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaDynamoDBExecutionRole in the AWS Managed Policy Reference Guide.

**AWS managed policy: AWSLambdaENIManagementAccess**

This policy grants permissions to create, describe, and delete elastic network interfaces used by a VPC-enabled Lambda function.

You can attach the AWSLambdaENIManagementAccess policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaENIManagementAccess in the AWS Managed Policy Reference Guide.

**AWS managed policy: AWSLambdaExecute**

This policy grants PUT and GET access to Amazon Simple Storage Service and full access to CloudWatch Logs.

You can attach the AWSLambdaExecute policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaExecute in the AWS Managed Policy Reference Guide.

**AWS managed policy: AWSLambdaInvocation-DynamoDB**

This policy grants read access to Amazon DynamoDB Streams.

You can attach the AWSLambdaInvocation-DynamoDB policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaInvocation-DynamoDB in the AWS Managed Policy Reference Guide.

**AWS managed policy: AWSLambdaKinesisExecutionRole**

This policy grants permissions to read events from an Amazon Kinesis data stream and write to CloudWatch Logs.

You can attach the AWSLambdaKinesisExecutionRole policy to your users, groups, and roles.
For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaKinesisExecutionRole in the AWS Managed Policy Reference Guide.

**AWS managed policy: AWSLambdaMSKExecutionRole**

This policy grants permissions to read and access records from an Amazon Managed Streaming for Apache Kafka cluster, manage elastic network interfaces, and write to CloudWatch Logs.

You can attach the AWSLambdaMSKExecutionRole policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaMSKExecutionRole in the AWS Managed Policy Reference Guide.

**AWS managed policy: AWSLambdaRole**

This policy grants permissions to invoke Lambda functions.

You can attach the AWSLambdaRole policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaRole in the AWS Managed Policy Reference Guide.

**AWS managed policy: AWSLambdaSQSQueueExecutionRole**

This policy grants permissions to read and delete messages from an Amazon Simple Queue Service queue, and grants write permissions to CloudWatch Logs.

You can attach the AWSLambdaSQSQueueExecutionRole policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaSQSQueueExecutionRole in the AWS Managed Policy Reference Guide.

**AWS managed policy: AWSLambdaVPCAccessExecutionRole**

This policy grants permissions to manage elastic network interfaces within an Amazon Virtual Private Cloud and write to CloudWatch Logs.

You can attach the AWSLambdaVPCAccessExecutionRole policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaVPCAccessExecutionRole in the AWS Managed Policy Reference Guide.
Lambda updates to AWS managed policies

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AWSLambda_ReadOnlyAccess</strong> – Change</td>
<td>Lambda updated the AWSLambda_ReadOnlyAccess policy to allow principals to list AWS CloudFormation stacks.</td>
<td>July 27, 2023</td>
</tr>
<tr>
<td>AWS Lambda started tracking changes</td>
<td>AWS Lambda started tracking changes for its AWS managed policies.</td>
<td>July 27, 2023</td>
</tr>
</tbody>
</table>

Troubleshooting AWS Lambda identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Lambda and IAM.

Topics

- I am not authorized to perform an action in Lambda
- I am not authorized to perform iam:PassRole
- I'm an administrator and want to migrate from AWS managed policies for Lambda that will be deprecated
- I want to allow people outside of my AWS account to access my Lambda resources

I am not authorized to perform an action in Lambda

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your sign-in credentials.

The following example error occurs when the mateojackson user tries to use the console to view details about a function but does not have lambda:GetFunction permissions.
In this case, Mateo asks his administrator to update his policies to allow him to access the my-function resource using the `lambda:GetFunction` action.

### I am not authorized to perform `iam:PassRole`

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Lambda.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Lambda. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

### I'm an administrator and want to migrate from AWS managed policies for Lambda that will be deprecated

After March 1, 2021, the AWS managed policies `AWSLambdaReadOnlyAccess` and `AWSLambdaFullAccess` will be deprecated and can no longer be attached to new users. For more information about policy deprecations, see [Deprecated AWS managed policies](https://docs.aws.amazon.com/iam/latest/userguide/deprecated-policies.html) in the *IAM User Guide*.

Lambda has introduced two new AWS managed policies:

- The `AWSLambda_ReadOnlyAccess` policy grants read-only access to Lambda, Lambda console features, and other related AWS services. This policy was created by scoping down the previous policy `AWSLambdaReadOnlyAccess`.
The **AWSLambda_FullAccess** policy grants full access to Lambda, Lambda console features, and other related AWS services. This policy was created by scoping down the previous policy **AWSLambdaFullAccess**.

### Using the AWS managed policies

We recommend using the newly launched managed policies to grant users, groups, and roles access to Lambda; however, review the permissions granted in the policies to ensure they meet your requirements.

- To review the permissions of the **AWSLambda_ReadOnlyAccess** policy, see the [AWSLambda_ReadOnlyAccess](#) policy page in the IAM console.
- To review the permissions of the **AWSLambda_FullAccess** policy, see the [AWSLambda_FullAccess](#) policy page in the IAM console.

After reviewing the permissions, you can attach the policies to an IAM identity (groups, users, or roles). For instructions about attaching an AWS managed policy, see [Adding and removing IAM identity permissions](#) in the IAM User Guide.

### Using customer managed policies

If you need more fine-grained access control or would like to add permissions, you can create your own customer managed policies. For more information, see [Creating policies on the JSON tab](#) in the IAM User Guide.

### I want to allow people outside of my AWS account to access my Lambda resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Lambda supports these features, see [How AWS Lambda works with IAM](#).
- To learn how to provide access to your resources across AWS accounts that you own, see [Providing access to an IAM user in another AWS account that you own](#) in the IAM User Guide.
• To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the IAM User Guide.

• To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the IAM User Guide.

• To learn the difference between using roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the IAM User Guide.

Governance for AWS Lambda

To build and deploy serverless, cloud-native applications, you must allow for agility and speed to market with appropriate governance and guardrails. You set business-level priorities, maybe emphasizing agility as the top priority, or alternatively emphasizing risk aversion via governance, guardrails, and controls. Realistically, you won't have an "either/or" strategy but an "and" strategy that balances both agility and guardrails in your software development lifecycle. No matter where these requirements fall in your company's lifecycle, governance capabilities are likely to become an implementation requirement in your processes and toolchains.

Here are a few examples of governance controls that an organization might implement for Lambda:

• Lambda functions must not be publicly accessible.

• Lambda functions must be attached to a VPC.

• Lambda functions should not use deprecated runtimes.

• Lambda functions must be tagged with a set of required tags.

• Lambda layers must not be accessible outside of the organization.

• Lambda functions with an attached security group must have matching tags between the function and security group.

• Lambda functions with an attached layer must use an approved version

• Lambda environment variables must be encrypted at rest with a customer managed key.

The following diagram is an example of an in-depth governance strategy that implements controls and policy throughout the software development and deployment process:
The following topics explain how to implement controls for developing and deploying Lambda functions in your organization, both for the startup and the enterprise. Your organization might already have tools in place. The following topics take a modular approach to these controls, so that you can pick and choose the components you actually need.

Topics

- Proactive controls for Lambda with AWS CloudFormation Guard
- Proactive controls for Lambda with AWS Config
- Detective controls for Lambda with AWS Config
- Lambda code signing with AWS Signer
- Lambda code scanning with Amazon Inspector
- Implement observability for Lambda security and compliance
Proactive controls for Lambda with AWS CloudFormation Guard

AWS CloudFormation Guard is an open-source, general-purpose, policy-as-code evaluation tool. This can be used for preventative governance and compliance by validating Infrastructure as Code (IaC) templates and service compositions against policy rules. These rules can be customized based on your team or organizational requirements. For Lambda functions, the Guard rules can be used to control resource creation and configuration updates by defining the required property settings needed while creating or updating a Lambda function.

Compliance administrators define the list of controls and governance policies that are required for deploying and updating Lambda functions. Platform administrators implement the controls in CI/CD pipelines, as pre-commit validation webhooks with code repositories, and provide developers with command line tools for validating templates and code on local workstations. Developers author code, validate templates with command line tools, and then commit code to repositories, which are then automatically validated via the CI/CD pipelines prior to deployment into an AWS environment.

Guard allows you to write your rules and implement your controls with a domain-specific language as follows.

For example, suppose you want to ensure that developers choose only the latest runtimes. You could specify two different policies, one to identify runtimes that are already deprecated and another to identify runtimes that are to be deprecated soon. To do this, you might write the following etc/rules.guard file:

Sample Rule:  
AWS::EC2::Volume Encrypted != false << you must create encrypted volumes


For example, suppose you want to ensure that developers choose only the latest runtimes. You could specify two different policies, one to identify runtimes that are already deprecated and another to identify runtimes that are to be deprecated soon. To do this, you might write the following etc/rules.guard file:
let lambda_functions = Resources.*[
    Type == "AWS::Lambda::Function"
]

rule lambda_already_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
                Runtime !in ["dotnetcore3.1", "nodejs12.x", "python3.6", "python2.7", "dotnet5.0", "dotnetcore2.1", "ruby2.5", "nodejs10.x", "nodejs8.10", "nodejs4.3", "nodejs6.10", "dotnetcore1.0", "dotnetcore2.0", "nodejs4.3-edge", "nodejs"] <<Lambda function is using a deprecated runtime.>>
            }
        }
    }
}

rule lambda_soon_to_be_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
                Runtime !in ["nodejs16.x", "nodejs14.x", "python3.7", "java8", "dotnet7", "go1.x", "ruby2.7", "provided"] <<Lambda function is using a runtime that is targeted for deprecation.>>
            }
        }
    }
}

Now suppose you write the following iac/lambda.yaml CloudFormation template that defines a Lambda function:

```yaml
Fn:
  Type: AWS::Lambda::Function
  Properties:
    Runtime: python3.7
    CodeUri: src
    Handler: fn.handler
    Role: !GetAtt FnRole.Arn
  Layers:
```
After **installing** the Guard utility, validate your template:

```bash
cfn-guard validate --rules etc/rules.guard --data iac/lambda.yaml
```

The output looks like this:

```
lambda.yaml Status = FAIL
FAILED rules
rules.guard/lambda_soon_to_be_deprecated_runtime
---
Evaluating data lambda.yaml against rules rules.guard
Number of non-compliant resources 1
Resource = Fn {
    Type = AWS::Lambda::Function
    Rule = lambda_soon_to_be_deprecated_runtime {
        ALL {
            Check = Runtime not IN
            "nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"] {
                ComparisonError {
                    Message = Lambda function is using a runtime that is targeted for deprecation.
                    Error = Check was not compliant as property [/Resources/Fn/Properties/Runtime[L:88,C:15]] was not present in [(resolved, Path=[L:0,C:0] Value=["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"])]
                }
            PropertyPath = /Resources/Fn/Properties/Runtime[L:88,C:15]
            Operator = NOT IN
            Value = "python3.7"
            ComparedWith =
                
                "nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]
        }
    }
}
```
Guard allows your developers to see from their local developer workstations that they need to update the template to use a runtime that is allowed by the organization. This happens prior to committing to a code repository and subsequently failing checks within a CI/CD pipeline. As a result, your developers get this feedback on how to develop compliant templates and shift their time to writing code that delivers business value. This control can be applied on the local developer workstation, in a pre-commit validation webhook, and/or in the CI/CD pipeline prior to deployment.

**Caveats**

If you're using AWS Serverless Application Model (AWS SAM) templates to define Lambda functions, be aware that you need to update the Guard rule to search for the `AWS::Serverless::Function` resource type as follows.

```javascript
let lambda_functions = Resources.*[
    Type == "AWS::Serverless::Function"
]
```

Guard also expects the properties to be included within the resource definition. Meanwhile, AWS SAM templates allow for properties to be specified in a separate `Globals` section. Properties that are defined in the Globals section are not validated with your Guard rules.

As outlined in the Guard troubleshooting documentation, be aware that Guard doesn't support short-form intrinsics like `!GetAtt` or `!Sub` and instead requires using the expanded forms: `Fn::GetAtt` and `Fn::Sub`. (The earlier example doesn't evaluate the Role property, so the short-form intrinsic was used for simplicity.)
Proactive controls for Lambda with AWS Config

It is essential to ensure compliance in your serverless applications as early in the development process as possible. In this topic, we cover how to implement preventative controls using AWS Config. This allows you to implement compliance checks earlier in the development process and enables you to implement the same controls in your CI/CD pipelines. This also standardizes your controls in a centrally managed repository of rules so that you can apply your controls consistently across your AWS accounts.

For example, suppose your compliance administrators defined a requirement to ensure that all Lambda functions include AWS X-Ray tracing. With AWS Config’s proactive mode, you can run compliance checks on your Lambda function resources before deployment, reducing the risk of deploying improperly configured Lambda functions and saving developers time by giving them faster feedback on infrastructure as code templates. The following is a visualization of the flow for preventative controls with AWS Config:

Consider a requirement that all Lambda functions must have tracing enabled. In response, the platform team identifies the need for a specific AWS Config rule to run proactively across all
accounts. This rule flags any Lambda function that lacks a configured X-Ray tracing configuration as a non-compliant resource. The team develops a rule, packages it in a conformance pack, and deploys the conformance pack across all AWS accounts to ensure that all accounts in the organization uniformly apply these controls. You can write the rule in AWS CloudFormation Guard 2.x.x syntax, which takes the following form:

```
rule name when condition { assertion }
```

The following is a sample Guard rule that checks to ensure Lambda functions has tracing enabled:

```
rule lambda_tracing_check {
  when configuration.tracingConfig exists {
    configuration.tracingConfig.mode == "Active"
  }
}
```

The platform team takes further action by mandating that every AWS CloudFormation deployment invokes a pre-create/update hook. They assume full responsibility for developing this hook and configuring the pipeline, strengthening the centralized control of compliance rules and sustaining their consistent application across all deployments. To develop, package, and register a hook, see Walkthrough: Python in the CloudFormation Command Line Interface (CFN-CLI) documentation. You can use the CloudFormation CLI to create the hook project:

```
cfn init
```

This command asks you for some basic information about your hook project and creates a project with following files in it:

```
README.md
<hook-name>.json
rpk.log
src/handler.py
template.yml
hook-role.yaml
```

As a hook developer, you need to add the desired target resource type in the <hook-name>.json configuration file. In the configuration below, a hook is configured to execute before any Lambda function is created using CloudFormation. You can add similar handlers for preUpdate and preDelete actions as well.
"handlers": {
    "preCreate": {
        "targetNames": [
            "AWS::Lambda::Function"
        ],
        "permissions": []
    }
}

You also need to ensure that the CloudFormation hook has appropriate permissions to call the AWS Config APIs. You can do that by updating the role definition file named `hook-role.yaml`. The role definition file has the following trust policy by default, which allows CloudFormation to assume the role.

```
AssumeRolePolicyDocument:
  Version: '2012-10-17'
  Statement:
    - Effect: Allow
      Principal:
        Service:
        - hooks.cloudformation.amazonaws.com
        - resources.cloudformation.amazonaws.com
```

To allow this hook to call config APIs, you must add following permissions to the Policy statement. Then you submit the hook project using the `cfn submit` command, where CloudFormation creates a role for you with the required permissions.

```
Policies:
  - PolicyName: HookTypePolicy
    PolicyDocument:
      Version: '2012-10-17'
      Statement:
        - Effect: Allow
          Action:
            - "config:Describe*"
            - "config:Get*"
            - "config:List*"
            - "config:SelectResourceConfig"
          Resource: "*"
```
Next, you need to write a Lambda function in a `src/handler.py` file. Within this file, you find methods named `preCreate`, `preUpdate`, and `preDelete` already created when you initiated the project. You aim to write a common, reusable function that calls the AWS Config StartResourceEvaluation API in proactive mode using the AWS SDK for Python (Boto3). This API call takes resource properties as input and evaluates the resource against the rule definition.

```python
def validate_lambda_tracing_config(resource_type, function_properties: MutableMapping[str, Any]) -> ProgressEvent:
    LOG.info("Fetching proactive data")
    config_client = boto3.client('config')
    resource_specs = {
        'ResourceId': 'MyFunction',
        'ResourceType': resource_type,
        'ResourceConfiguration': json.dumps(function_properties),
        'ResourceConfigurationSchemaType': 'CFN_RESOURCE_SCHEMA'
    }
    LOG.info("Resource Specifications:", resource_specs)
    eval_response = config_client.start_resource_evaluation(EvaluationMode='PROACTIVE',
                                                             ResourceDetails=resource_specs,
                                                             EvaluationTimeout=60)
    ResourceEvaluationId = eval_response.ResourceEvaluationId
    compliance_response =
    config_client.get_compliance_details_by_resource(ResourceEvaluationId=ResourceEvaluationId)
    LOG.info("Compliance Verification:",
             compliance_response.EvaluationResults[0].ComplianceType)
    if "NON_COMPLIANT" ==
    compliance_response.EvaluationResults[0].ComplianceType:
        return ProgressEvent(status=OperationStatus.FAILED,
                              message="Lambda function found with no tracing enabled : FAILED",
                              errorCode=HandlerErrorCode.NonCompliant)
    else:
        return ProgressEvent(status=OperationStatus.SUCCESS,
                              message="Lambda function found with tracing enabled : PASS.")
```  

Now you can call the common function from the handler for the pre-create hook. Here's an example of the handler:

```python
@hook.handler(HookInvocationPoint.CREATE_PRE_PROVISION)
def pre_create_handler(
    session: Optional[SessionProxy],
    request: HookHandlerRequest,
    callback_context: MutableMapping[str, Any],
    type_configuration: TypeConfigurationModel
) -> ProgressEvent:
    LOG.info("Starting execution of the hook")
```
After this step you can register the hook and configure it to listen to all AWS Lambda function creation events.

A developer prepares the infrastructure as code (IaC) template for a serverless microservice using Lambda. This preparation includes adherence to internal standards, followed by locally testing and committing the template to the repository. Here's an example IaC template:

```
MyLambdaFunction:
  Type: 'AWS::Lambda::Function'
  Properties:
    Handler: index.handler
    Role: !GetAtt LambdaExecutionRole.Arn
    FunctionName: MyLambdaFunction
    Code:
      ZipFile: |
        import json
        
        def handler(event, context):
            return {
                'statusCode': 200,
                'body': json.dumps('Hello World!')
            }
    Runtime: python3.8
    TracingConfig:
      Mode: PassThrough
    MemorySize: 256
    Timeout: 10
```

As part of the CI/CD process, when the CloudFormation template is deployed, the CloudFormation service invokes the pre-create/update hook right before provisioning AWS::Lambda::Function resource type. The hook utilizes AWS Config rules running in proactive mode to verify that the Lambda function configuration includes the mandated tracing configuration. The response from the hook determines the next step. If compliant, the hook signals success, and CloudFormation...
proceeds to provision the resources. If not, the CloudFormation stack deployment fails, the pipeline comes to an immediate halt, and the system records the details for subsequent review. Compliance notifications are sent to the relevant stakeholders.

You can find the hook success/fail information in the CloudFormation console:

If you have logs enabled for your CloudFormation hook, you can capture the hook evaluation result. Here is a sample log for a hook with a failed status, indicating that the Lambda function does not have X-Ray enabled:

```
2023-09-29T23:58:17.574-05:00  ProgressEvent(status=OperationStatus.FAILED, 'FAILED>', errorCode='HandlerError', NonCompliant='NonCompliant', message='Lambda function found with no tracing enabled : FAILED', result=None, callbackContext=None, callbackDelaySeconds=0, resourceModel=None, resourceModels=None, nextToken=None)
```

...
If the developer chooses to change the IaC to update TracingConfig Mode value to Active and redeploy, the hook executes successfully and the stack proceeds with creating the Lambda resource.

In this way, you can implement preventative controls with AWS Config in proactive mode when developing and deploying serverless resources in your AWS accounts. By integrating AWS Config rules into the CI/CD pipeline, you can identify and optionally block non-compliant resource deployments, such as Lambda functions that lack an active tracing configuration. This ensures that only resources that comply with the latest governance policies are deployed into your AWS environments.
Detective controls for Lambda with AWS Config

In addition to proactive evaluation, AWS Config can also reactively detect resource deployments and configurations that do not comply with your governance policies. This is important because governance policies evolve as your organization learns and implements new best practices.

Consider a scenario where you set a brand new policy when deploying or updating Lambda functions: All Lambda functions must always use a specific, approved Lambda layer version. You can configure AWS Config to monitor new or updated functions for layer configurations. If AWS Config detects a function that is not using an approved layer version, it flags the function as a non-compliant resource. You can optionally configure AWS Config to automatically remediate the resource by specifying a remediation action using an AWS Systems Manager automation document. For example, you could write an automation document in Python using the AWS SDK for Python (Boto3), which updates the non-compliant function to point to the approved layer version. Thus, AWS Config serves as both a detective and corrective control, automating compliance management.

Let's break down this process into three important implementation phases:

Existing AWS Accounts

Phase 1: Identify access resources

Start by activating AWS Config across your accounts and configuring it to record AWS Lambda functions. This allows AWS Config to observe when Lambda functions are created or updated. You can then configure custom policy rules to check for specific policy violations, which use AWS CloudFormation Guard syntax. Guard rules take the following general form:
Below is a sample rule that checks to ensure that a layer is not set to an old layer version:

```java
rule desiredlayer when configuration.layers !empty {
    some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn
}
```

Let's understand the rule syntax and structure:

- **Rule name:** The name of the rule in the provided example is `desiredlayer`.
- **Condition:** This clause specifies the condition under which the rule should be checked. In the provided example, the condition is `configuration.layers !empty`. This means the resource should be evaluated only when the `layers` property in the configuration isn't empty.
- **Assertion:** After the `when` clause, an assertion determines what the rule checks. The assertion `some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn` checks if any of the Lambda layer ARNs do not match the `OldLayerArn` value. If they do not match, the assertion is true and the rule passes; otherwise, it fails.

`CONFIG_RULE_PARAMETERS` is a special set of parameters that is configured with the AWS Config rule. In this case, `OldLayerArn` is a parameter inside `CONFIG_RULE_PARAMETERS`. This allows users to provide a specific ARN value that they consider old or deprecated, and then the rule checks if any Lambda functions are using this old ARN.

**Phase 2: Visualize and design**

AWS Config gathers configuration data and stores that data in Amazon Simple Storage Service (Amazon S3) buckets. You can use [Amazon Athena](https://aws.amazon.com/athena/) to query this data directly from your S3 buckets. With Athena, you can aggregate this data at the organizational level, generating a holistic view of your resource configurations across all your accounts. To set up aggregation of resource configuration data, see [Visualizing AWS Config data using Athena and Amazon QuickSight](https://aws.amazon.com/blogs/cloud operations-and-management/visualizing-aws-config-data-using-athena-and-amazon-quicksight/) on the AWS Cloud Operations and Management blog.

The following is a sample Athena query to identify all Lambda functions using a particular layer ARN:

```sql
WITH unnested AS (  
```
**Here are results from the query:**

<table>
<thead>
<tr>
<th>Region</th>
<th>FunctionName</th>
<th>memory_size</th>
<th>timeout</th>
<th>version</th>
</tr>
</thead>
<tbody>
<tr>
<td>us-east-1</td>
<td>UPDATE.GetUserPullRequests</td>
<td>120</td>
<td>10</td>
<td>LATEST</td>
</tr>
<tr>
<td>us-east-1</td>
<td>Scheduler.Ci-Instance.ScheduledJob</td>
<td>120</td>
<td>10</td>
<td>LATEST</td>
</tr>
<tr>
<td>us-east-1</td>
<td>my_functions.function10</td>
<td>120</td>
<td>10</td>
<td>LATEST</td>
</tr>
<tr>
<td>us-east-1</td>
<td>lw-web-ui-CognitoIdentityAPI-UserPoolFunction-1TS0R9H0J6VYQD</td>
<td>120</td>
<td>10</td>
<td>LATEST</td>
</tr>
<tr>
<td>us-east-1</td>
<td>GetLatestApiKey</td>
<td>120</td>
<td>10</td>
<td>LATEST</td>
</tr>
<tr>
<td>us-east-1</td>
<td>aws-python-http-api-project-dev-hello</td>
<td>1004</td>
<td>6</td>
<td>LATEST</td>
</tr>
<tr>
<td>us-east-1</td>
<td>cloud-virtual-remote assessment</td>
<td>120</td>
<td>10</td>
<td>LATEST</td>
</tr>
<tr>
<td>us-east-1</td>
<td>my_functions.function1</td>
<td>120</td>
<td>10</td>
<td>LATEST</td>
</tr>
<tr>
<td>us-east-1</td>
<td>my_functions.function25</td>
<td>120</td>
<td>10</td>
<td>LATEST</td>
</tr>
</tbody>
</table>
With the AWS Config data aggregated across the organization, you can then create a dashboard using Amazon QuickSight. By importing your Athena results into Amazon QuickSight, you can visualize how well your Lambda functions adhere to the layer version rule. This dashboard can highlight compliant and non-compliant resources, which helps you to determine your enforcement policy, as outlined in the next section. The following image is an example dashboard that reports on the distribution of layer versions applied to functions within the organization.

Example Amazon QuickSight dashboard shows distribution of layer versions in Lambda functions.

**Phase 3: Implement and enforce**

You can now optionally pair your layer version rule that you created in phase 1 with a remediation action via a Systems Manager automation document, which you author as a Python script written with AWS SDK for Python (Boto3). The script calls the **UpdateFunctionConfiguration** API action for each Lambda function, updating the function configuration with the new layer ARN. Alternatively, you could have the script submit a pull request to the code repository to update the layer ARN. This way future code deployments are also updated with the correct layer ARN.
Lambda code signing with AWS Signer

AWS Signer is a fully managed code-signing service that allows you to validate your code against a digital signature to confirm that code is unaltered and from a trusted publisher. AWS Signer can be used in conjunction with AWS Lambda to verify that functions and layers are unaltered prior to deployment into your AWS environments. This protects your organization from malicious actors who might have gained credentials to create new or update existing functions.

To set up code signing for your Lambda functions, start by creating an S3 bucket with versioning enabled. After that, create a signing profile with AWS Signer, specify Lambda as the platform and then specify a period of days in which the signing profile is valid. Example:

```yaml
Signer:
  Type: AWS::Signer::SigningProfile
  Properties:
    PlatformId: AWSLambda-SHA384-ECDSA
    SignatureValidityPeriod:
      Type: DAYS
      Value: !Ref pValidDays
```

Then use the signing profile and create a signing configuration with Lambda. You have to specify what to do when the signing configuration sees an artifact that does not match a digital signature that it expected: warn (but allow the deployment) or enforce (and block the deployment). The example below is configured to enforce and block deployments.

```yaml
SigningConfig:
  Type: AWS::Lambda::CodeSigningConfig
  Properties:
    AllowedPublishers:
      SigningProfileVersionArns:
        - !GetAtt Signer.ProfileVersionArn
    CodeSigningPolicies:
      UntrustedArtifactOnDeployment: Enforce
```

You now have AWS Signer configured with Lambda to block untrusted deployments. Let's assume you've finished coding a feature request and are now ready to deploy the function. The first step is to zip the code up with the appropriate dependencies and then sign the artifact using the signing profile that you created. You can do this by uploading the zip artifact to the S3 bucket and then starting a signing job.
aws signer start-signing-job \
--source 's3={bucketName=your-versioned-bucket,key=your-prefix/your-zip-artifact.zip,version=QyaJ3c4qa50LXV.9VaZgXHlsGbvcXxpT}' \
--destination 's3={bucketName=your-versioned-bucket,prefix=your-prefix/}' \
--profile-name your-signer-id

You get an output as follows, where the jobId is the object that is created in the destination bucket and prefix and jobOwner is the 12-digit AWS account ID where the job was run.

```
{
    "jobId": "87a3522b-5c0b-4d7d-b4e0-4255a8e05388",
    "jobOwner": "111122223333"
}
```

And now you can deploy your function using the signed S3 object and the code signing configuration that you created.

```
Fn:
    Type: AWS::Serverless::Function
    Properties:
        CodeUri: s3://your-versioned-bucket/your-prefix/87a3522b-5c0b-4d7d-b4e0-4255a8e05388.zip
        Handler: fn.handler
        Role: !GetAtt FnRole.Arn
        CodeSigningConfigArn: !Ref pSigningConfigArn
```

You can alternatively test a function deployment with the original unsigned source zip artifact. The deployment should fail with the following message:

```
Lambda cannot deploy the function. The function or layer might be signed using a signature that the client is not configured to accept. Check the provided signature for unsigned.
```

If you are building and deploying your functions using the AWS Serverless Application Model (AWS SAM), the package command handles uploading the zip artifact to S3 and also starts the signing job and gets the signed artifact. You can do this with the following command and parameters:

```
sam package -t your-template.yaml \
--output-template-file your-output.yaml \
--s3-bucket your-versioned-bucket \
```
AWS Signer helps you verify that zip artifacts that are deployed into your accounts are trusted for deployment. You can include the process above in your CI/CD pipelines and require that all functions have a code signing configuration attached using the techniques outlined in previous topics. By using code signing with your Lambda function deployments, you prevent malicious actors who might have gotten credentials to create or update functions from injecting malicious code in your functions.

```
--s3-prefix your-prefix \
--signing-profiles your-signer-id
```
**Lambda code scanning with Amazon Inspector**

Amazon Inspector is a vulnerability management service that continually scans workloads for known software vulnerabilities and unintended network exposure. Amazon Inspector creates a finding that describes the vulnerability, identifies the affected resource, rates the severity of the vulnerability, and provides remediation guidance.

Amazon Inspector support provides continuous, automated security vulnerability assessments for Lambda functions and layers. Amazon Inspector provides two scan types for Lambda:

- **Lambda standard scanning (default):** Scans application dependencies within a Lambda function and its layers for package vulnerabilities.
- **Lambda code scanning:** Scans the custom application code in your functions and layers for code vulnerabilities. You can either activate Lambda standard scanning or activate Lambda standard scanning together with Lambda code scanning.

To enable Amazon Inspector, navigate to the Amazon Inspector console, expand the Settings section, and choose Account Management. On the Accounts tab, choose Activate, and then select one of the scan options.

You can enable Amazon Inspector for multiple accounts and delegate permissions to manage Amazon Inspector for the organization to specific accounts while setting up Amazon Inspector. While enabling, you need to grant Amazon Inspector permissions by creating the role: AWSServiceRoleForAmazonInspector2. The Amazon Inspector console allows you to create this role using a one-click option.

For Lambda standard scanning, Amazon Inspector initiates vulnerability scans of Lambda functions in the following situations:

- As soon as Amazon Inspector discovers an existing Lambda function.
- When you deploy a new Lambda function.
- When you deploy an update to the application code or dependencies of an existing Lambda function or its layers.
- Whenever Amazon Inspector adds a new common vulnerabilities and exposures (CVE) item to its database, and that CVE is relevant to your function.
For Lambda code scanning, Amazon Inspector evaluates your Lambda function application code using automated reasoning and machine learning that analyzes your application code for overall security compliance. If Amazon Inspector detects a vulnerability in your Lambda function application code, Amazon Inspector produces a detailed **Code Vulnerability** finding. For a list of possible detections, see the [Amazon CodeGuru Detector Library](#).

To view the findings, go to the [Amazon Inspector console](#). On the **Findings** menu, choose **By Lambda function** to display the security scan results that were performed on Lambda functions.

To exclude a Lambda function from standard scanning, tag the function with the following key-value pair:

- **Key:** InspectorExclusion
- **Value:** LambdaStandardScanning

To exclude a Lambda function from code scans, tag the function with the following key-value pair:

- **Key:** InspectorCodeExclusion
- **Value:** LambdaCodeScanning

For example, as shown in following image, Amazon Inspector automatically detects vulnerabilities and categorizes the findings of type **Code Vulnerability**, which indicates that the vulnerability is in the code of the function, and not in one of the code-dependent libraries. You can check these details for a specific function or multiple functions at once.
You can dive further into each of these findings and learn how to remediate the issue.
Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior or failure of the Lambda function.

**Finding overview**

<table>
<thead>
<tr>
<th>AWS account ID</th>
<th>[Hiden]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Severity</td>
<td>High</td>
</tr>
<tr>
<td>Type</td>
<td>Code Vulnerability</td>
</tr>
<tr>
<td>Detector name</td>
<td>Override of reserved variable names in a Lambda function</td>
</tr>
<tr>
<td>Relevant CWE</td>
<td>--</td>
</tr>
<tr>
<td>Rule ID</td>
<td>Rule-434311</td>
</tr>
<tr>
<td>Detector tags</td>
<td>#availability, #aws-python-sdk, #aws-lambda, #data-integrity, #maintainability, #security, #security-context, #python</td>
</tr>
<tr>
<td>Fix available</td>
<td>Yes</td>
</tr>
<tr>
<td>Created at</td>
<td>March 29, 2023 10:08 AM (UTC-04:00)</td>
</tr>
</tbody>
</table>

**Vulnerability details**

- **File path**: lambda_function.py

**Vulnerability location**

```python
3 import socket
4 
5 def lambda_handler(event, context):
6 
7     # print("Scenario 1");
8     os.environ['_HANDLER'] = 'hello'
9     # print("Scenario 1 ends")
10 
11     # print("Scenario 2");
12     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13     s.bind(('0',0))
```

**Suggested remediation**

Your code attempts to override an environment variable that is reserved by the Lambda runtime environment. This can lead to unexpected behavior and might break the execution of your Lambda function.
While working with your Lambda functions, ensure that you comply with the naming conventions for your Lambda functions. For more information, see Using Lambda environment variables.

You are responsible for the remediation suggestions that you accept. Always review remediation suggestions before accepting them. You might need to make edits to remediation suggestions to ensure that your code does what you intended.
Implement observability for Lambda security and compliance

AWS Config is a useful tool to find and fix non-compliant AWS Serverless resources. Every change you make to your serverless resources is recorded in AWS Config. Additionally, AWS Config allows you to store configuration snapshot data on S3. You can use Amazon Athena and Amazon QuickSight to make dashboards and see AWS Config data. In Detective controls for Lambda with AWS Config, we discussed how we can visualize a certain configuration like Lambda layers. This topic expands on these concepts.

Visibility into Lambda configurations

You can use queries to pull important configurations like AWS account ID, Region, AWS X-Ray tracing configuration, VPC configuration, memory size, runtime, and tags. Here is a sample query you can use to pull this information from Athena:

```sql
WITH unnested AS (
    SELECT
        item.awsaccountid AS account_id,
        item.awsregion AS region,
        item.configuration AS lambda_configuration,
        item.resourceid AS resourceid,
        item.resourcename AS resourcename,
        item.configuration AS configuration,
        json_parse(item.configuration) AS lambda_json
    FROM
        default.aws_config_configuration_snapshot,
        UNNEST(configurationitems) as t(item)
    WHERE
        "dt" = 'latest'
        AND item.resourcetype = 'AWS::Lambda::Function'
)

SELECT DISTINCT
    account_id,
    tags,
    region as Region,
    resourcename as FunctionName,
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.runtime') AS version
    json_extract_scalar(lambda_json, '$.vpcConfig.SubnetIds') AS vpcConfig
    json_extract_scalar(lambda_json, '$.tracingConfig.mode') AS tracingConfig
```
You can use the query to build an Amazon QuickSight dashboard and visualize the data. To aggregate AWS resource configuration data, create tables in Athena, and build Amazon QuickSight dashboards on the data from Athena, see Visualizing AWS Config data using Athena and Amazon QuickSight on the AWS Cloud Operations and Management blog. Notably, this query also retrieves tag information for the functions. This allows for deeper insights into your workloads and environments, especially if you employ custom tags.

For more information on actions that you can take, see the Addressing the observability findings section later in this topic.

Visibility into Lambda compliance

With the data generated by AWS Config, you can create organization-level dashboards to monitor compliance. This allows for consistent tracking and monitoring of:

- Compliance packs by compliance score
- Rules by non-compliant resources
Check each rule to identify non-compliant resources for that rule. For example, if your organization mandates that all Lambda functions must be associated with a VPC and if you have deployed an AWS Config rule to identify compliance, you can select the `lambda-inside-vpc` rule in the list above.
For more information on actions that you can take, see the Addressing the observability findings section below.

Visibility into Lambda function boundaries using Security Hub

To ensure that AWS services including Lambda are used securely, AWS introduced the Foundational Security Best Practices v1.0.0. This set of best practices provides clear guidelines for securing resources and data in the AWS environment, emphasizing the importance of maintaining a strong security posture. The AWS Security Hub complements this by offering a unified security and compliance center. It aggregates, organizes, and prioritizes security findings from multiple
AWS services like Amazon Inspector, AWS Identity and Access Management Access Analyzer, and Amazon GuardDuty.

If you have Security Hub, Amazon Inspector, IAM Access Analyzer, and GuardDuty enabled within your AWS organization, Security Hub automatically aggregates findings from these services. For instance, let's consider Amazon Inspector. Using Security Hub, you can efficiently identify code and package vulnerabilities in Lambda functions. In the Security Hub console, navigate to the bottom section labeled **Latest findings from AWS integrations.** Here, you can view and analyze findings sourced from various integrated AWS services.

<table>
<thead>
<tr>
<th>Latest findings from AWS integrations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon GuardDuty</td>
</tr>
<tr>
<td>Open the GuardDuty console</td>
</tr>
<tr>
<td>Amazon Inspector</td>
</tr>
<tr>
<td>Open the Inspector console</td>
</tr>
<tr>
<td>Amazon Macie</td>
</tr>
<tr>
<td>Open the Macie console</td>
</tr>
<tr>
<td>AWS Health</td>
</tr>
<tr>
<td>Open the Personal Health Dashboard</td>
</tr>
<tr>
<td>AWS IAM Access Analyzer</td>
</tr>
<tr>
<td>Open the IAM Access Analyzer console</td>
</tr>
<tr>
<td>AWS Systems Manager Patch Manager</td>
</tr>
<tr>
<td>Open the Systems Manager Patch Manager console</td>
</tr>
<tr>
<td>AWS Firewall Manager</td>
</tr>
<tr>
<td>Open the Firewall Manager console</td>
</tr>
</tbody>
</table>

To see details, choose the **See findings** link in the second column. This displays a list of findings filtered by product, such as Amazon Inspector. To limit your search to Lambda functions, set `ResourceType` to `AwsLambdaFunction`. This displays findings from Amazon Inspector related to Lambda functions.
For GuardDuty, you can identify suspicious network traffic patterns. Such anomalies might suggest the existence of potentially malicious code within your Lambda function.

With IAM Access Analyzer, you can check policies, especially those with condition statements that grant function access to external entities. Moreover, IAM Access Analyzer evaluates permissions set when using the `AddPermission` operation in the Lambda API alongside an `EventSourceToken`.

**Addressing the observability findings**

Given the wide-ranging configurations possible for Lambda functions and their distinct requirements, a standardized automation solution for remediation might not suit every situation. Additionally, changes are implemented differently across various environments. If you encounter any configuration that seems non-compliant, consider the following guidelines:

1. **Tagging strategy**

   We recommend implementing a comprehensive tagging strategy. Each Lambda function should be tagged with key information such as:

   - **Owner**: The person or team responsible for the function.
   - **Environment**: Production, staging, development, or sandbox.
   - **Application**: The broader context to which this function belongs, if applicable.

2. **Owner outreach**
Instead of automating the breaking changes (like VPC configuration adjustment), proactively contact the owners of non-compliant functions (identified by the owner tag) providing them sufficient time to either:

- Adjust non-compliant configurations on Lambda functions.
- Provide an explanation and request an exception, or refine the compliance standards.

3. **Maintain a configuration management database (CMDB)**

While tags can provide immediate context, maintaining a centralized CMDB can provide deeper insights. It can hold more granular information about each Lambda function, its dependencies, and other critical metadata. A CMDB is an invaluable resource for auditing, compliance checks, and identifying function owners.

As the landscape of serverless infrastructure continually evolves, it's essential to adopt a proactive stance towards monitoring. With tools like AWS Config, Security Hub, and Amazon Inspector, potential anomalies or non-compliant configurations can be swiftly identified. However, tools alone cannot ensure total compliance or optimal configurations. It's crucial to pair these tools with well-documented processes and best practices.

- **Feedback loop:** Once remediation steps are undertaken, ensure there's a feedback loop. This means periodically revisiting non-compliant resources to confirm if they've been updated or are still running with the same issues.

- **Documentation:** Always document the observations, actions taken, and any exceptions granted. Proper documentation not only helps during audits but also aids in enhancing the process for better compliance and security in the future.

- **Training and awareness:** Ensure that all stakeholders, especially Lambda function owners, are regularly trained and made aware of best practices, organizational policies, and compliance mandates. Regular workshops, webinars, or training sessions can go a long way in ensuring everyone is on the same page when it comes to security and compliance.

In conclusion, while tools and technologies provide robust capabilities to detect and flag potential issues, the human element—understanding, communication, training, and documentation—remains pivotal. Together, they form a potent combination to ensure that your Lambda functions and broader infrastructure remain compliant, secure, and optimized for your business needs.
Compliance validation for AWS Lambda

Third-party auditors assess the security and compliance of AWS Lambda as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others.

For a list of AWS services in scope of specific compliance programs, see AWS services in scope by compliance program. For general information, see AWS compliance programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading reports in AWS artifact.

Your compliance responsibility when using Lambda is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. You can implement governance controls to ensure that your company's Lambda functions meet your compliance requirements. For more information, see Governance for AWS Lambda.

Resilience in AWS Lambda

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS global infrastructure.

In addition to the AWS global infrastructure, Lambda offers several features to help support your data resiliency and backup needs.

- **Versioning** – You can use versioning in Lambda to save your function's code and configuration as you develop it. Together with aliases, you can use versioning to perform blue/green and rolling deployments. For details, see Lambda function versions.

- **Scaling** – When your function receives a request while it's processing a previous request, Lambda launches another instance of your function to handle the increased load. Lambda automatically scales to handle 1,000 concurrent executions per Region, a quota that can be increased if needed. For details, see Lambda function scaling.

- **High availability** – Lambda runs your function in multiple Availability Zones to ensure that it is available to process events in case of a service interruption in a single zone. If you configure your
function to connect to a virtual private cloud (VPC) in your account, specify subnets in multiple Availability Zones to ensure high availability. For details, see Connecting outbound networking to resources in a VPC.

- **Reserved concurrency** – To make sure that your function can always scale to handle additional requests, you can reserve concurrency for it. Setting reserved concurrency for a function ensures that it can scale to, but not exceed, a specified number of concurrent invocations. This ensures that you don't lose requests due to other functions consuming all of the available concurrency. For details, see Configuring reserved concurrency.

- **Retries** – For asynchronous invocations and a subset of invocations triggered by other services, Lambda automatically retries on error with delays between retries. Other clients and AWS services that invoke functions synchronously are responsible for performing retries. For details, see Error handling and automatic retries in AWS Lambda.

- **Dead-letter queue** – For asynchronous invocations, you can configure Lambda to send requests to a dead-letter queue if all retries fail. A dead-letter queue is an Amazon SNS topic or Amazon SQS queue that receives events for troubleshooting or reprocessing. For details, see Dead-letter queues.

**Infrastructure security in AWS Lambda**

As a managed service, AWS Lambda is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in Security Pillar AWS Well-Architected Framework.

You use AWS published API calls to access Lambda through the network. Clients must support the following:

- **Transport Layer Security (TLS).** We require TLS 1.2 and recommend TLS 1.3.

- **Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman).** Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.
Monitoring and troubleshooting Lambda functions

AWS Lambda integrates with other AWS services to help you monitor and troubleshoot your Lambda functions. Lambda automatically monitors Lambda functions on your behalf and reports metrics through Amazon CloudWatch. To help you monitor your code when it runs, Lambda automatically tracks the number of requests, the invocation duration per request, and the number of requests that result in an error.

You can use other AWS services to troubleshoot your Lambda functions. This section describes how to use these AWS services to monitor, trace, debug, and troubleshoot your Lambda functions and applications. For details about function logging and errors in each runtime, see individual runtime sections.

For more information about monitoring Lambda applications, see Monitoring and observability in Serverless Land.

Sections

- Monitoring functions on the Lambda console
- Working with Lambda function metrics
- Using Amazon CloudWatch logs with AWS Lambda
- Using AWS Lambda with AWS X-Ray
- Using Lambda Insights in Amazon CloudWatch
- Using CodeGuru Profiler with your Lambda function
- Example workflows using other AWS services
Monitoring functions on the Lambda console

The Lambda service monitors functions on your behalf and sends metrics to Amazon CloudWatch. The Lambda console creates monitoring graphs for these metrics and shows them on the Monitoring page for each Lambda function.

The Lambda console provides a single pane view of metrics, logs, and traces. The console provides filters for time range, time zone, and refresh options that apply to all panes universally. You can easily correlate metrics, logs, and traces, reducing the mean time to recovery (MTTR) when troubleshooting errors in your Lambda functions.

Pricing

CloudWatch has a perpetual free tier. Beyond the free tier threshold, CloudWatch charges for metrics, dashboards, alarms, logs, and insights. For more information, see Amazon CloudWatch pricing.

Using the Lambda console

You can monitor your Lambda functions and applications on the Lambda console.

To monitor a function

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose the Monitor tab.

Types of monitoring graphs

The following section describes the monitoring graphs on the Lambda console.

Lambda monitoring graphs

- **Invocations** – The number of times that the function was invoked.
- **Duration** – The average, minimum, and maximum amount of time your function code spends processing an event.
- **Error count and success rate (%)** – The number of errors and the percentage of invocations that completed without error.
- **Throttles** – The number of times that an invocation failed due to concurrency limits.
- **IteratorAge** – For stream event sources, the age of the last item in the batch when Lambda received it and invoked the function.
- **Async delivery failures** – The number of errors that occurred when Lambda attempted to write to a destination or dead-letter queue.
- **Concurrent executions** – The number of function instances that are processing events.

**Viewing graphs on the Lambda console**

The following section describes how to view CloudWatch monitoring graphs on the Lambda console, and open the CloudWatch metrics dashboard.

**To view monitoring graphs for a function**

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose the **Monitor** tab.
4. Choose from the predefined time ranges, or choose a custom time range.
5. To see the definition of a graph in CloudWatch, choose the three vertical dots (**Widget actions**), and then choose **View in metrics** to open the **Metrics** dashboard on the CloudWatch console.
Viewing queries on the CloudWatch Logs console

The following section describes how to view and add reports from CloudWatch Logs Insights to a custom dashboard on the CloudWatch Logs console.

**To view reports for a function**

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose the Monitor tab.
4. Choose View logs in CloudWatch.
5. Choose View in Logs Insights.
6. Choose from the predefined time ranges, or choose a custom time range.
7. Choose Run query.
8. (Optional) Choose Save.
What's next?

- Learn about the metrics that Lambda records and sends to CloudWatch in Working with Lambda function metrics.
- Learn how to use CloudWatch Lambda Insights to collect and aggregate Lambda function runtime performance metrics and logs in Using Lambda Insights in Amazon CloudWatch.
Working with Lambda function metrics

When your AWS Lambda function finishes processing an event, Lambda sends metrics about the invocation to Amazon CloudWatch. There is no charge for these metrics.

On the CloudWatch console, you can build graphs and dashboards with these metrics. You can set alarms to respond to changes in utilization, performance, or error rates. Lambda sends metric data to CloudWatch in 1-minute intervals. For more immediate insight into your Lambda function, you can create high-resolution custom metrics as described in Serverless Land. Charges apply for custom metrics and CloudWatch alarms. For more information, see Amazon CloudWatch Pricing.

This page describes the Lambda function invocation, performance, and concurrency metrics available on the CloudWatch console.

Sections
- Viewing metrics on the CloudWatch console
- Types of metrics

Viewing metrics on the CloudWatch console

You can use the CloudWatch console to filter and sort function metrics by function name, alias, or version.

To view metrics on the CloudWatch console

1. Open the Metrics page (AWS/Lambda namespace) of the CloudWatch console.
2. On the Browse tab, under Metrics, choose any of the following dimensions:

   - **By Function Name** (FunctionName) – View aggregate metrics for all versions and aliases of a function.
   - **By Resource** (Resource) – View metrics for a version or alias of a function.
   - **By Executed Version** (ExecutedVersion) – View metrics for a combination of alias and version. Use the ExecutedVersion dimension to compare error rates for two versions of a function that are both targets of a weighted alias.
   - **Across All Functions** (none) – View aggregate metrics for all functions in the current AWS Region.
3. Choose a metric, then choose Add to graph or another graphing option.
By default, graphs use the Sum statistic for all metrics. To choose a different statistic and customize the graph, use the options on the Graphed metrics tab.

**Note**
The timestamp on a metric reflects when the function was invoked. Depending on the duration of the invocation, this can be several minutes before the metric is emitted. For example, if your function has a 10-minute timeout, then look more than 10 minutes in the past for accurate metrics.

For more information about CloudWatch, see the [Amazon CloudWatch User Guide](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/).  

**Types of metrics**

The following section describes the types of Lambda metrics available on the CloudWatch console.

**Invocation metrics**

Invocation metrics are binary indicators of the outcome of a Lambda function invocation. For example, if the function returns an error, then Lambda sends the Errors metric with a value of 1. To get a count of the number of function errors that occurred each minute, view the Sum of the Errors metric with a period of 1 minute.

**Note**
View the following invocation metrics with the Sum statistic.

- **Invocations** – The number of times that your function code is invoked, including successful invocations and invocations that result in a function error. Invocations aren't recorded if the invocation request is throttled or otherwise results in an invocation error. The value of Invocations equals the number of requests billed.
- **Errors** – The number of invocations that result in a function error. Function errors include exceptions that your code throws and exceptions that the Lambda runtime throws. The runtime returns errors for issues such as timeouts and configuration errors. To calculate the error rate, divide the value of Errors by the value of Invocations. Note that the timestamp on an error metric reflects when the function was invoked, not when the error occurred.
• **DeadLetterErrors** – For asynchronous invocation, the number of times that Lambda attempts to send an event to a dead-letter queue (DLQ) but fails. Dead-letter errors can occur due to misconfigured resources or size limits.

• **DestinationDeliveryFailures** – For asynchronous invocation and supported event source mappings, the number of times that Lambda attempts to send an event to a destination but fails. For event source mappings, Lambda supports destinations for stream sources (DynamoDB and Kinesis). Delivery errors can occur due to permissions errors, misconfigured resources, or size limits. Errors can also occur if the destination you have configured is an unsupported type such as an Amazon SQS FIFO queue or an Amazon SNS FIFO topic.

• **Throttles** – The number of invocation requests that are throttled. When all function instances are processing requests and no concurrency is available to scale up, Lambda rejects additional requests with a TooManyRequestsException error. Throttled requests and other invocation errors don't count as either Invocations or Errors.

• **OversizedRecordCount** – For Amazon DocumentDB event sources, the number of events your function receives from your change stream that are over 6 MB in size. Lambda drops the message and emits this metric.

• **ProvisionedConcurrencyInvocations** – The number of times that your function code is invoked using provisioned concurrency.

• **ProvisionedConcurrencySpilloverInvocations** – The number of times that your function code is invoked using standard concurrency when all provisioned concurrency is in use.

• **RecursiveInvocationsDropped** – The number of times that Lambda has stopped invocation of your function because it's detected that your function is part of an infinite recursive loop. Lambda recursive loop detection monitors how many times a function is invoked as part of a chain of requests by tracking metadata added by supported AWS SDKs. If your function is invoked as part of a chain of requests more than 16 times, Lambda drops the next invocation.

**Performance metrics**

Performance metrics provide performance details about a single function invocation. For example, the Duration metric indicates the amount of time in milliseconds that your function spends processing an event. To get a sense of how fast your function processes events, view these metrics with the Average or Max statistic.
• **Duration** – The amount of time that your function code spends processing an event. The billed duration for an invocation is the value of **Duration** rounded up to the nearest millisecond. **Duration** does not include cold start time.

• **PostRuntimeExtensionsDuration** – The cumulative amount of time that the runtime spends running code for extensions after the function code has completed.

• **IteratorAge** – For DynamoDB, Kinesis, and Amazon DocumentDB event sources, the age of the last record in the event. This metric measures the time between when a stream receives the record and when the event source mapping sends the event to the function.

• **OffsetLag** – For self-managed Apache Kafka and Amazon Managed Streaming for Apache Kafka (Amazon MSK) event sources, the difference in offset between the last record written to a topic and the last record that your function's consumer group processed. Though a Kafka topic can have multiple partitions, this metric measures the offset lag at the topic level.

**Duration** also supports percentile (p) statistics. Use percentiles to exclude outlier values that skew Average and Maximum statistics. For example, the p95 statistic shows the maximum duration of 95 percent of invocations, excluding the slowest 5 percent. For more information, see **Percentiles** in the *Amazon CloudWatch User Guide*.

### Concurrency metrics

Lambda reports concurrency metrics as an aggregate count of the number of instances processing events across a function, version, alias, or AWS Region. To see how close you are to hitting concurrency limits, view these metrics with the Max statistic.

• **ConcurrentExecutions** – The number of function instances that are processing events. If this number reaches your **concurrent executions quota** for the Region, or the **reserved concurrency** limit on the function, then Lambda throttles additional invocation requests.

• **ProvisionedConcurrentExecutions** – The number of function instances that are processing events using **provisioned concurrency**. For each invocation of an alias or version with provisioned concurrency, Lambda emits the current count.

• **ProvisionedConcurrencyUtilization** – For a version or alias, the value of ProvisionedConcurrentExecutions divided by the total amount of provisioned concurrency allocated. For example, .5 indicates that 50 percent of allocated provisioned concurrency is in use.

• **UnreservedConcurrentExecutions** – For a Region, the number of events that functions without reserved concurrency are processing.
• **ClaimedAccountConcurrency** – For a Region, the amount of concurrency that is unavailable for on-demand invocations. ClaimedAccountConcurrency is equal to UnreservedConcurrentExecutions plus the amount of allocated concurrency (i.e. the total reserved concurrency plus total provisioned concurrency). For more information, see [Working with the ClaimedAccountConcurrency metric](#).

### Asynchronous invocation metrics

Asynchronous invocation metrics provide details about asynchronous invocations from event sources and direct invocations. You can set thresholds and alarms to notify you of certain changes. For example, when there’s an undesired increase in the number of events queued for processing (AsyncEventsReceived). Or, when an event has been waiting a long time to be processed (AsyncEventAge).

• **AsyncEventsReceived** – The number of events that Lambda successfully queues for processing. This metric provides insight into the number of events that a Lambda function receives. Monitor this metric and set alarms for thresholds to check for issues. For example, to detect an undesirable number of events sent to Lambda, and to quickly diagnose issues resulting from incorrect trigger or function configurations. Mismatches between AsyncEventsReceived and Invocations can indicate a disparity in processing, events being dropped, or a potential queue backlog.

• **AsyncEventAge** – The time between when Lambda successfully queues the event and when the function is invoked. The value of this metric increases when events are being retried due to invocation failures or throttling. Monitor this metric and set alarms for thresholds on different statistics for when a queue buildup occurs. To troubleshoot an increase in this metric, look at the Errors metric to identify function errors and the Throttles metric to identify concurrency issues.

• **AsyncEventsDropped** – The number of events that are dropped without successfully executing the function. If you configure a dead-letter queue (DLQ) or OnFailure destination, then events are sent there before they're dropped. Events are dropped for various reasons. For example, events can exceed the maximum event age or exhaust the maximum retry attempts, or reserved concurrency might be set to 0. To troubleshoot why events are dropped, look at the Errors metric to identify function errors and the Throttles metric to identify concurrency issues.
Using Amazon CloudWatch logs with AWS Lambda

AWS Lambda automatically monitors Lambda functions on your behalf to help you troubleshoot failures in your functions. As long as your function's execution role has the necessary permissions, Lambda captures logs for all requests handled by your function and sends them to Amazon CloudWatch Logs.

You can insert logging statements into your code to help you validate that your code is working as expected. Lambda automatically integrates with CloudWatch Logs and sends all logs from your code to a CloudWatch logs group associated with a Lambda function.

By default, Lambda sends logs to a log group named /aws/lambda/<function name>. If you want your function to send logs to another group, you can configure this using the Lambda console, the AWS Command Line Interface (AWS CLI) or the Lambda API. See the section called “Configuring CloudWatch log groups” to learn more.

You can view logs for Lambda functions using the Lambda console, the CloudWatch console, the AWS Command Line Interface (AWS CLI), or the CloudWatch API.

Note
It may take 5 to 10 minutes for logs to show up after a function invocation.

Section
- Prerequisites
- Pricing
- Configuring advanced logging controls for your Lambda function
- Accessing logs with the Lambda console
- Accessing logs with the AWS CLI
- Runtime function logging
- What's next?
Prerequisites

Your execution role needs permission to upload logs to CloudWatch Logs. You can add CloudWatch Logs permissions using the AWSLambdaBasicExecutionRole AWS managed policy provided by Lambda. To add this policy to your role, run the following command:

```
aws iam attach-role-policy --role-name your-role --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

For more information, see AWS managed policies for Lambda features.

Pricing

There is no additional charge for using Lambda logs; however, standard CloudWatch Logs charges apply. For more information, see CloudWatch pricing.

Configuring advanced logging controls for your Lambda function

To give you more control over how your functions’ logs are captured, processed, and consumed, Lambda offers the following logging configuration options:

- **Log format** - select between plain text and structured JSON format for your function’s logs
- **Log level** - for JSON structured logs, choose the detail level of the logs Lambda sends to CloudWatch, such as ERROR, DEBUG, or INFO
- **Log group** - choose the CloudWatch log group your function sends logs to

Configuring JSON and plain text log formats

Capturing your log outputs as JSON key value pairs makes it easier to search and filter when debugging your functions. With JSON formatted logs, you can also add tags and contextual information to your logs. This can help you to perform automated analysis of large volumes of log data. Unless your development workflow relies on existing tooling that consumes Lambda logs in plain text, we recommend that you select JSON for your log format.

For all Lambda managed runtimes, you can choose whether your function's system logs are sent to CloudWatch Logs in unstructured plain text or JSON format. System logs are the logs that Lambda generates and are sometimes known as platform event logs.
For **supported runtimes**, when you use one of the supported built-in logging methods, Lambda can also output your function's application logs (the logs your function code generates) in structured JSON format. When you configure your function's log format for these runtimes, the configuration you choose applies to both system and application logs.

For supported runtimes, if your function uses a supported logging library or method, you don't need to make any changes to your existing code for Lambda to capture logs in structured JSON.

**Note**

Using JSON log formatting adds additional metadata and encodes log messages as JSON objects containing a series of key value pairs. Because of this, the size of your function's log messages can increase.

**Supported runtimes and logging methods**

Lambda currently supports the option to output JSON structured application logs for the following runtimes.

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Supported versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>All Java runtimes except Java 8 on Amazon Linux 1</td>
</tr>
<tr>
<td>Node.js</td>
<td>Node.js 16 and later</td>
</tr>
<tr>
<td>Python</td>
<td>Python 3.7 and later</td>
</tr>
</tbody>
</table>

For Lambda to send your function's application logs to CloudWatch in structured JSON format, your function must use the following built-in logging tools to output logs:

- **Java** - the `LambdaLogger` logger or Log4j2.
- **Node.js** - The console methods `console.trace`, `console.debug`, `console.log`, `console.info`, `console.error`, and `console.warn`
- **Python** - the standard Python logging library
For more information about using advanced logging controls with supported runtimes, see the section called “Logging”, the section called “Logging”, and the section called “Logging”.

For other managed Lambda runtimes, Lambda currently only natively supports capturing system logs in structured JSON format. However, you can still capture application logs in structured JSON format in any runtime by using logging tools such as Powertools for AWS Lambda that output JSON formatted log outputs.

**Default log formats**

Currently, the default log format for all Lambda runtimes is plain text.

If you’re already using logging libraries like Powertools for AWS Lambda to generate your function logs in JSON structured format, you don’t need to change your code if you select JSON log formatting. Lambda doesn’t double-encode any logs that are already JSON encoded, so your function’s application logs will continue to be captured as before.

**JSON format for system logs**

When you configure your function's log format as JSON, each system log item (platform event) is captured as a JSON object that contains key value pairs with the following keys:

- "time" - the time the log message was generated
- "level" - the log level assigned to the message
- "type" - the type of event being logged
- "record" - the contents of the log output

The format of the "record" value varies according to the type of event being logged. For more information see the section called “Telemetry API Event object types”. For more information about the log levels assigned to system log events, see the section called “System log level event mapping”.

For comparison, the following two examples show the same log output in both plain text and structured JSON formats. Note that in most cases, system log events contain more information when output in JSON format than when output in plain text.
Example plain text:

2023-03-13 18:56:24.046000 fbe8c1 INIT_START Runtime Version:
python:3.9.v18 Runtime Version ARN: arn:aws:lambda:euro-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5e6baaf619cd0fc0

Example structured JSON:

```json
{
    "time": "2023-03-13T18:56:24.046Z",
    "level": "INFO",
    "type": "platform.initStart",
    "record": {
        "initializationType": "on-demand",
        "phase": "init",
        "runtimeVersion": "python:3.9.v18",
        "runtimeVersionArn": "arn:aws:lambda:euro-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5e6baaf619cd0fc0"
    }
}
```

Note

The section called “Telemetry API” always emits platform events such as START and REPORT in JSON format. Configuring the format of the system logs Lambda sends to CloudWatch doesn’t affect Lambda Telemetry API behavior.

JSON format for application logs

When you configure your function's log format as JSON, application log outputs written using supported logging libraries and methods are captured as a JSON object that contains key value pairs with the following keys.

- "timestamp" - the time the log message was generated
- "level" - the log level assigned to the message
- "message" - the contents of the log message
- "requestId" (Python and Node.js) or "AWSRequestId" (Java) - the unique request ID for the function invocation
Depending on the runtime and logging method that your function uses, this JSON object may also contain additional key pairs. For example, in Node.js, if your function uses console methods to log error objects using multiple arguments, The JSON object will contain extra key value pairs with the keys `errorMessage`, `errorType`, and `stackTrace`. To learn more about JSON formatted logs in different Lambda runtimes, see the section called “Logging”, the section called “Logging”, and the section called “Logging”.

Note

The key Lambda uses for the timestamp value is different for system logs and application logs. For system logs, Lambda uses the key "time" to maintain consistency with Telemetry API. For application logs, Lambda follows the conventions of the supported runtimes and uses "timestamp".

For comparison, the following two examples show the same log output in both plain text and structured JSON formats.

Example plain text:

```
2023-10-27T19:17:45.586Z 79b4f56e-95b1-4643-9700-2807f4e68189 INFO some log message
```

Example structured JSON:

```
{
    "timestamp":"2023-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"some log message",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

Setting your function’s log format

To configure the log format for your function, you can use the Lambda console or the AWS Command Line Interface (AWS CLI). You can also configure a function’s log format using the `CreateFunction` and `UpdateFunctionConfiguration` Lambda API commands, the AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` resource, and the AWS CloudFormation `AWS::Lambda::Function` resource.
Changing your function's log format doesn’t affect existing logs stored in CloudWatch Logs. Only new logs will use the updated format.

If you change your function's log format to JSON and do not set log level, then Lambda automatically sets your function's application log level and system log level to INFO. This means that Lambda sends only log outputs of level INFO and lower to CloudWatch Logs. To learn more about application and system log-level filtering see the section called “Log-level filtering”.

**Note**

For Python runtimes, when your function's log format is set to plain text, the default log-level setting is WARN. This means that Lambda only sends log outputs of level WARN and lower to CloudWatch Logs. Changing your function's log format to JSON changes this default behavior. To learn more about logging in Python, see the section called “Logging”.

For Node.js functions that emit embedded metric format (EMF) logs, changing your function's log format to JSON could result in CloudWatch being unable to recognize your metrics.

**Important**

If your function uses Powertools for AWS Lambda (TypeScript) or the open-sourced EMF client libraries to emit EMF logs, update your Powertools and EMF libraries to the latest versions to ensure that CloudWatch can continue to parse your logs correctly. If you switch to the JSON log format, we also recommend that you carry out testing to ensure compatibility with your function's embedded metrics. For further advice about node.js functions that emit EMF logs, see the section called “Using embedded metric format (EMF) client libraries with structured JSON logs”.

**To configure a function's log format (console)**

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. On the function configuration page, choose Monitoring and operations tools.
4. In the Logging configuration pane, choose Edit.
5. Under Log content, for Log format select either Text or JSON.
6. Choose **Save**.

**To change the log format of an existing function (AWS CLI)**

- To change the log format of an existing function, use the `update-function-configuration` command. Set the `LogFormat` option in `LoggingConfig` to either `JSON` or `Text`.

  ```bash
  aws lambda update-function-configuration \
  --function-name myFunction --logging-config LogFormat=JSON
  ```

**To set log format when you create a function (AWS CLI)**

- To configure log format when you create a new function, use the `--logging-config` option in the `create-function` command. Set `LogFormat` to either `JSON` or `Text`. The following example command creates a function using the Node.js 18 runtime that outputs logs in structured JSON.

  ```bash
  aws lambda create-function --function-name myFunction --runtime nodejs18.x \
  --handler index.handler --zip-file fileb://function.zip \
  --role arn:aws:iam::123456789012:role/LambdaRole --logging-config LogFormat=JSON
  ```

**Log-level filtering**

Lambda can filter your function's logs so that only logs of a certain detail level or lower are sent to CloudWatch Logs. You can configure log-level filtering separately for your function's system logs (the logs that Lambda generates) and application logs (the logs that your function code generates).

For the section called “Supported runtimes and logging methods”, you don't need to make any changes to your function code for Lambda to filter your function's application logs.
For all other runtimes and logging methods, your function code must output log events to stdout or stderr as JSON formatted objects that contain a key value pair with the key "level". For example, Lambda interprets the following output to stdout as a DEBUG level log.

```python
print({'"level": "debug", "msg": "my debug log", "timestamp": "2023-11-02T16:51:31.587199Z"'})
```

If the "level" value field is invalid or missing, Lambda will assign the log output the level INFO. For Lambda to use the timestamp field, you must specify the time in valid RFC 3339 timestamp format. If you don't supply a valid timestamp, Lambda will assign the log the level INFO and add a timestamp for you.

When naming the timestamp key, follow the conventions of the runtime you are using. Lambda supports most common naming conventions used by the managed runtimes. For example, in functions that use the .NET runtime, Lambda recognizes the key "Timestamp".

**Note**

To use log-level filtering, your function must be configured to use the JSON log format. The default log format for all Lambda managed runtimes is currently plain text. To learn how to configure your function's log format to JSON, see the section called “Setting your function's log format”.

For application logs (the logs generated by your function code), you can choose between the following log levels.

<table>
<thead>
<tr>
<th>Log level</th>
<th>Standard usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACE (most detail)</td>
<td>The most fine-grained information used to trace the path of your code's execution</td>
</tr>
<tr>
<td>DEBUG</td>
<td>Detailed information for system debugging</td>
</tr>
<tr>
<td>INFO</td>
<td>Messages that record the normal operation of your function</td>
</tr>
<tr>
<td>WARN</td>
<td>Messages about potential errors that may lead to unexpected behavior if unaddressed</td>
</tr>
</tbody>
</table>
When you select a log level, Lambda sends logs at that level and lower to CloudWatch Logs. For example, if you set a function’s application log level to WARN, Lambda doesn’t send log outputs at the INFO and DEBUG levels. The default application log level for log filtering is INFO.

When Lambda filters your function’s application logs, log messages with no level will be assigned the log level INFO.

For system logs (the logs generated by the Lambda service), you can choose between the following log levels.

<table>
<thead>
<tr>
<th>Log level</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG (most detail)</td>
<td>Detailed information for system debugging</td>
</tr>
<tr>
<td>INFO</td>
<td>Messages that record the normal operation of your function</td>
</tr>
<tr>
<td>WARN (least detail)</td>
<td>Messages about potential errors that may lead to unexpected behavior if unaddressed</td>
</tr>
</tbody>
</table>

When you select a log level, Lambda sends logs at that level and lower. For example, if you set a function’s system log level to INFO, Lambda doesn’t send log outputs at the DEBUG level.

By default, Lambda sets the system log level to INFO. With this setting, Lambda automatically sends "start" and "report" log messages to CloudWatch. To receive more or less detailed system logs, change the log level to DEBUG or WARN. To see a list of the log levels that Lambda maps different system log events to, see the section called “System log level event mapping”.

Configuring advanced logging controls for your Lambda function
Configuring log-level filtering

To configure application and system log-level filtering for your function, you can use the Lambda console or the AWS Command Line Interface (AWS CLI). You can also configure a function’s log level using the `CreateFunction` and `UpdateFunctionConfiguration` Lambda API commands, the AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` resource, and the AWS CloudFormation `AWS::Lambda::Function` resource.

Note that if you set your function's log level in your code, this setting takes precedence over any other log level settings you configure. For example, if you use the Python `logging` `setLevel()` method to set your function’s logging level to INFO, this setting takes precedence over a setting of WARN that you configure using the Lambda console.

To configure an existing function’s application or system log level (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. On the function configuration page, choose Monitoring and operations tools.
4. In the Logging configuration pane, choose Edit.
5. Under Log content, for Log format ensure JSON is selected.
6. Using the radio buttons, select your desired Application log level and System log level for your function.
7. Choose Save.

To configure an existing function’s application or system log level (AWS CLI)

- To change the application or system log level of an existing function, use the `update-function-configuration` command. Set `--system-log-level` to one of DEBUG, INFO, or WARN. Set `--application-log-level` to one of DEBUG, INFO, WARN, ERROR, or FATAL.

```
aws lambda update-function-configuration \
  --function-name myFunction --system-log-level WARN \
  --application-log-level ERROR
```
To configure log-level filtering when you create a function

- To configure log-level filtering when you create a new function, use the `--system-log-level` and `--application-log-level` options in the `create-function` command. Set `--system-log-level` to one of DEBUG, INFO, or WARN. Set `--application-log-level` to one of DEBUG, INFO, WARN, WARN, or FATAL.

```
aws lambda create-function --function-name myFunction --runtime nodejs18.x \
  --handler index.handler --zip-file fileb://function.zip \
  --role arn:aws:iam::123456789012:role/LambdaRole --system-log-level WARN \
  --application-log-level ERROR
```

System log level event mapping

For system level log events generated by Lambda, the following table defines the log level assigned to each event. To learn more about the events listed in the table, see the section called "Event schema reference".

<table>
<thead>
<tr>
<th>Event name</th>
<th>Condition</th>
<th>Assigned log level</th>
</tr>
</thead>
<tbody>
<tr>
<td>initStart</td>
<td>runtimeVersion is set</td>
<td>INFO</td>
</tr>
<tr>
<td>initStart</td>
<td>runtimeVersion is not set</td>
<td>DEBUG</td>
</tr>
<tr>
<td>initRuntimeDone</td>
<td>status=success</td>
<td>DEBUG</td>
</tr>
<tr>
<td>initRuntimeDone</td>
<td>status!=success</td>
<td>WARN</td>
</tr>
<tr>
<td>initReport</td>
<td>initializationType=snapstart</td>
<td>INFO</td>
</tr>
<tr>
<td>initReport</td>
<td>initializationType!=snapstart</td>
<td>DEBUG</td>
</tr>
<tr>
<td>initReport</td>
<td>status!=success</td>
<td>WARN</td>
</tr>
<tr>
<td>restoreStart</td>
<td>runtimeVersion is set</td>
<td>INFO</td>
</tr>
<tr>
<td>restoreStart</td>
<td>runtimeVersion is not set</td>
<td>DEBUG</td>
</tr>
<tr>
<td>restoreRuntimeDone</td>
<td>status=success</td>
<td>DEBUG</td>
</tr>
<tr>
<td>Event name</td>
<td>Condition</td>
<td>Assigned log level</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>restoreRuntimeDone</td>
<td>status!=success</td>
<td>WARN</td>
</tr>
<tr>
<td>restoreReport</td>
<td>status=success</td>
<td>INFO</td>
</tr>
<tr>
<td>restoreReport</td>
<td>status=success</td>
<td>WARN</td>
</tr>
<tr>
<td>start</td>
<td>-</td>
<td>INFO</td>
</tr>
<tr>
<td>runtimeDone</td>
<td>status=success</td>
<td>DEBUG</td>
</tr>
<tr>
<td>runtimeDone</td>
<td>status!=success</td>
<td>WARN</td>
</tr>
<tr>
<td>report</td>
<td>status=success</td>
<td>INFO</td>
</tr>
<tr>
<td>report</td>
<td>status!=success</td>
<td>WARN</td>
</tr>
<tr>
<td>extension</td>
<td>state=success</td>
<td>INFO</td>
</tr>
<tr>
<td>extension</td>
<td>state!=success</td>
<td>WARN</td>
</tr>
<tr>
<td>logSubscription</td>
<td>-</td>
<td>INFO</td>
</tr>
<tr>
<td>telemetrySubscription</td>
<td>-</td>
<td>INFO</td>
</tr>
<tr>
<td>logsDropped</td>
<td>-</td>
<td>WARN</td>
</tr>
</tbody>
</table>

**Note**

The section called “Telemetry API” always emits the complete set of platform events. Configuring the level of the system logs Lambda sends to CloudWatch doesn't affect Lambda Telemetry API behavior.

**Application log-level filtering with custom runtimes**

When you configure application log-level filtering for your function, behind the scenes Lambda sets the application log level in the runtime using the AWS_LAMBDA_LOG_LEVEL environment variable. Lambda also sets your function's log format using the AWS_LAMBDA_LOG_FORMAT
environment variable. You can use these variables to integrate Lambda advanced logging controls into a [custom runtime](#).

For the ability to configure logging settings for a function using a custom runtime with the Lambda console, AWS CLI, and Lambda APIs, configure your custom runtime to check the value of these environment variables. You can then configure your runtime's loggers in accordance with the log format and log levels you select.

### Configuring CloudWatch log groups

By default, CloudWatch automatically creates a log group named /aws/lambda/<function name> for your function when it's first invoked. To configure your function to send logs to an existing log group, or to create a new log group for your function, you can use the Lambda console or the AWS CLI. You can also configure custom log groups using the [CreateFunction](#) and [UpdateFunctionConfiguration](#) Lambda API commands and the AWS Serverless Application Model (AWS SAM) [AWS::Serverless::Function](#) resource.

You can configure multiple Lambda functions to send logs to the same CloudWatch log group. For example, you could use a single log group to store logs for all of the Lambda functions that make up a particular application. When you use a custom log group for a Lambda function, the log streams Lambda creates include the function name and function version. This ensures that the mapping between log messages and functions is preserved, even if you use the same log group for multiple functions.

The naming format for custom log groups follows this convention:

```
YYYY/MM/DD/<function_name>[<function_version>][<execution_environment_GUID>]
```

Note that when configuring a custom log group, the name you select for your log group must follow the [CloudWatch Logs naming rules](#). Additionally, custom log group names mustn't begin with the string aws/. If you create a custom log group beginning with aws/, Lambda won't be able to create the log group. As a result of this, your function's logs won't be sent to CloudWatch.

### To change a function's log group (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. On the function configuration page, choose Monitoring and operations tools.
4. In the Logging configuration pane, choose Edit.
5. In the **Logging group** pane, for **CloudWatch log group**, choose **Custom**.

6. Under **Custom log group**, enter the name of the CloudWatch log group you want your function to send logs to. If you enter the name of an existing log group, then your function will use that group. If no log group exists with the name that you enter, then Lambda will create a new log group for your function with that name.

**To change a function's log group (AWS CLI)**

- To change the log group of an existing function, use the `update-function-configuration` command. If you specify the name of an existing log group, then your function will use that group. If no log group exists with the name that you specify, then Lambda will create a new log group for your function with that name.

  ```bash
  aws lambda update-function-configuration \n  --function-name myFunction --log-group myLogGroup
  ```

**To specify a custom log group when you create a function (AWS CLI)**

- To specify a custom log group when you create a new Lambda function using the AWS CLI, use the `--log-group` option. If you specify the name of an existing log group, then your function will use that group. If no log group exists with the name that you specify, then Lambda will create a new log group for your function with that name.

  The following example command creates a Node.js Lambda function that sends logs to a log group named `myLogGroup`.

  ```bash
  aws lambda create-function --function-name myFunction --runtime nodejs18.x \n  --handler index.handler --zip-file fileb://function.zip \n  --role arn:aws:iam::123456789012:role/LambdaRole --log-group myLogGroup
  ```

**Execution role permissions**

For your function to send logs to CloudWatch Logs, it must have the `logs:PutLogEvents` permission. When you configure your function's log group using the Lambda console, if your function doesn't have this permission, Lambda adds it to the function's **execution role** by default. When Lambda adds this permission, it gives the function permission to send logs to any CloudWatch Logs log group.
To prevent Lambda from automatically updating the function's execution role and edit it manually instead, expand Permissions and uncheck Add required permissions.

When you configure your function's log group using the AWS CLI, Lambda won't automatically add the logs:PutLogEvents permission. Add the permission to your function's execution role if it doesn't already have it. This permission is included in the AWSLambdaBasicExecutionRole managed policy.

**Accessing logs with the Lambda console**

To view logs using the Lambda console

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose Monitor.
4. Choose View logs in CloudWatch.

**Accessing logs with the AWS CLI**

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface (AWS CLI) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

You can use the [AWS CLI](#) to retrieve logs for an invocation using the --log-type command option. The response contains a LogResult field that contains up to 4 KB of base64-encoded logs from the invocation.

**Example retrieve a log ID**

The following example shows how to retrieve a log ID from the LogResult field for a function named my-function.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:
Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --decode
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-aws-configure.html) in the [AWS Command Line Interface User Guide for Version 2](https://docs.aws.amazon.com/cli/latest/userguide/)

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgO3b3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-592e5ca-f57281b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib,
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Duration: 79.67 ms Billed
Duration: 80 ms Memory Size: 128 MB Max Memory Used: 73 MB
```

The `base64` utility is available on Linux, macOS, and [Ubuntu on Windows](https://ubuntu.com), macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.
Copy the contents of the following code sample and save in your Lambda project directory as get-logs.sh.

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/index.html) in the *AWS Command Line Interface User Guide for Version 2*.

```bash
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-name stream1 --limit 5
```

**Example macOS and Linux (only)**

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```bash
chmod -R 755 get-logs.sh
```

**Example retrieve the last five log events**

In the same command prompt, run the following script to get the last five log events.

```bash
./get-logs.sh
```

You should see the following output:

```json
{
   "StatusCode": 200,
   "ExecutedVersion": "$LATEST"
}
```

```json
{
   "events": [
      {
         "timestamp": 1559763003171,
         "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST"
      },
```
Runtime function logging

To debug and validate that your code is working as expected, you can output logs with the standard logging functionality for your programming language. The Lambda runtime uploads your function's log output to CloudWatch Logs. For language-specific instructions, see the following topics:

- AWS Lambda function logging in Node.js
- AWS Lambda function logging in Python
- AWS Lambda function logging in Ruby
• AWS Lambda function logging in Java
• AWS Lambda function logging in Go
• Lambda function logging in C#
• AWS Lambda function logging in PowerShell

What's next?

• Learn more about log groups and accessing them through the CloudWatch console in Monitoring system, application, and custom log files in the Amazon CloudWatch User Guide.
Using AWS Lambda with AWS X-Ray

You can use AWS X-Ray to visualize the components of your application, identify performance bottlenecks, and troubleshoot requests that resulted in an error. Your Lambda functions send trace data to X-Ray, and X-Ray processes the data to generate a service map and searchable trace summaries.

If you've enabled X-Ray tracing in a service that invokes your function, Lambda sends traces to X-Ray automatically. The upstream service, such as Amazon API Gateway, or an application hosted on Amazon EC2 that is instrumented with the X-Ray SDK, samples incoming requests and adds a tracing header that tells Lambda to send traces or not. Traces from upstream message producers, such as Amazon SQS, are automatically linked to traces from downstream Lambda functions, creating an end-to-end view of the entire application. For more information, see Tracing event-driven applications in the AWS X-Ray Developer Guide.

Note

X-Ray tracing is currently not supported for Lambda functions with Amazon Managed Streaming for Apache Kafka (Amazon MSK), self-managed Apache Kafka, Amazon MQ with ActiveMQ and RabbitMQ, or Amazon DocumentDB event source mappings.

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose Configuration and then choose Monitoring and operations tools.
4. Choose Edit.
5. Under X-Ray, toggle on Active tracing.
6. Choose Save.
Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see AWS X-Ray pricing.

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's execution role. Otherwise, add the AWSXRayDaemonWriteAccess policy to the execution role.

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

Note

You cannot configure the X-Ray sampling rate for your functions.

In X-Ray, a trace records information about a request that is processed by one or more services. Services record segments that contain layers of subsegments. Lambda records a segment for the Lambda service that handles the invocation request, and one for the work done by the function. The function segment comes with subsegments for Initialization, Invocation, Restore (Lambda SnapStart only), and Overhead. For more information, see Lambda execution environment lifecycle.

Note

X-Ray treats unhandled exceptions in your Lambda function as Error statuses. X-Ray records Fault statuses only when Lambda experiences internal server errors. For more information, see Errors, faults, and exceptions in the X-Ray Developer Guide.

The Initialization subsegment represents the init phase of the Lambda execution environment lifecycle. During this phase, Lambda creates or unfreezes an execution environment with the resources you have configured, downloads the function code and all layers, initializes extensions, initializes the runtime, and runs the function's initialization code.
The Invocation subsegment represents the invoke phase where Lambda invokes the function handler. This begins with runtime and extension registration and it ends when the runtime is ready to send the response.

(Lambda SnapStart only) The Restore subsegment shows the time it takes for Lambda to restore a snapshot, load the runtime (JVM), and run any afterRestore runtime hooks. The process of restoring snapshots can include time spent on activities outside the MicroVM. This time is reported in the Restore subsegment. You aren't charged for the time spent outside the microVM to restore a snapshot.

The Overhead subsegment represents the phase that occurs between the time when the runtime sends the response and the signal for the next invoke. During this time, the runtime finishes all tasks related to an invoke and prepares to freeze the sandbox.

⚠️ Note
Occasionally, you may notice a large gap between the function initialization and invocation phases in your X-Ray traces. For functions using provisioned concurrency, this is because Lambda initializes your function instances well in advance of invocation. For functions using unreserved (on-demand) concurrency, Lambda may proactively initialize a function instance, even if there's no invocation. Visually, both of these cases show up as a time gap between the initialization and invocation phases.

⚠️ Important
In Lambda, you can use the X-Ray SDK to extend the Invocation subsegment with additional subsegments for downstream calls, annotations, and metadata. You can't access the function segment directly or record work done outside of the handler invocation scope.

See the following topics for a language-specific introduction to tracing in Lambda:

- Instrumenting Node.js code in AWS Lambda
- Instrumenting Python code in AWS Lambda
- Instrumenting Ruby code in AWS Lambda
- Instrumenting Java code in AWS Lambda
For a full list of services that support active instrumentation, see Supported AWS services in the AWS X-Ray Developer Guide.

Sections

- Execution role permissions
- The AWS X-Ray daemon
- Enabling active tracing with the Lambda API
- Enabling active tracing with AWS CloudFormation

Execution role permissions

Lambda needs the following permissions to send trace data to X-Ray. Add them to your function's execution role.

- xray:PutTraceSegments
- xray:PutTelemetryRecords

These permissions are included in the AWSXRayDaemonWriteAccess managed policy.

The AWS X-Ray daemon

Instead of sending trace data directly to the X-Ray API, the X-Ray SDK uses a daemon process. The AWS X-Ray daemon is an application that runs in the Lambda environment and listens for UDP traffic that contains segments and subsegments. It buffers incoming data and writes it to X-Ray in batches, reducing the processing and memory overhead required to trace invocations.

The Lambda runtime allows the daemon to up to 3 percent of your function's configured memory or 16 MB, whichever is greater. If your function runs out of memory during invocation, the runtime terminates the daemon process first to free up memory.

The daemon process is fully managed by Lambda and cannot be configured by the user. All segments generated by function invocations are recorded in the same account as the Lambda function. The daemon cannot be configured to redirect them to any other account.
For more information, see [The X-Ray daemon](https://docs.aws.amazon.com/xray/latest/userguide/what-is-xray.html) in the X-Ray Developer Guide.

## Enabling active tracing with the Lambda API

To manage tracing configuration with the AWS CLI or AWS SDK, use the following API operations:

- `UpdateFunctionConfiguration`
- `GetFunctionConfiguration`
- `CreateFunction`

The following example AWS CLI command enables active tracing on a function named *my-function*.

```bash
aws lambda update-function-configuration --function-name my-function \
  --tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

## Enabling active tracing with AWS CloudFormation

To activate tracing on an `AWS::Lambda::Function` resource in an AWS CloudFormation template, use the `TracingConfig` property.

### Example `function-inline.yml` – Tracing configuration

```yaml
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

For an AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` resource, use the `Tracing` property.

### Example `template.yml` – Tracing configuration

```yaml
Resources:
```
function:
    Type: AWS::Serverless::Function
Properties:
    Tracing: Active
...
Using Lambda Insights in Amazon CloudWatch

Amazon CloudWatch Lambda Insights collects and aggregates Lambda function runtime performance metrics and logs for your serverless applications. This page describes how to enable and use Lambda Insights to diagnose issues with your Lambda functions.

Sections

- How Lambda Insights monitors serverless applications
- Pricing
- Supported runtimes
- Enabling Lambda Insights in the Lambda console
- Enabling Lambda Insights programmatically
- Using the Lambda Insights dashboard
- Example workflow to detect function anomalies
- Example workflow using queries to troubleshoot a function
- What's next?

How Lambda Insights monitors serverless applications

CloudWatch Lambda Insights is a monitoring and troubleshooting solution for serverless applications running on AWS Lambda. The solution collects, aggregates, and summarizes system-level metrics including CPU time, memory, disk and network usage. It also collects, aggregates, and summarizes diagnostic information such as cold starts and Lambda worker shutdowns to help you isolate issues with your Lambda functions and resolve them quickly.

Lambda Insights uses a new CloudWatch Lambda Insights extension, which is provided as a Lambda layer. When you enable this extension on a Lambda function for a supported runtime, it collects system-level metrics and emits a single performance log event for every invocation of that Lambda function. CloudWatch uses embedded metric formatting to extract metrics from the log events. For more information, see Using AWS Lambda extensions.

The Lambda Insights layer extends the CreateLogStream and PutLogEvents for the /aws/lambda-insights/log group.
**Pricing**

When you enable Lambda Insights for your Lambda function, Lambda Insights reports 8 metrics per function and every function invocation sends about 1KB of log data to CloudWatch. You only pay for the metrics and logs reported for your function by Lambda Insights. There are no minimum fees or mandatory service usage policies. You do not pay for Lambda Insights if the function is not invoked. For a pricing example, see [Amazon CloudWatch pricing](https://aws.amazon.com/cloudwatch/pricing/).

**Supported runtimes**

You can use Lambda Insights with any of the runtimes that support [Lambda extensions](https://docs.aws.amazon.com/lambda/latest/dg/lambda-extended-runtimes.html).

**Enabling Lambda Insights in the Lambda console**

You can enable Lambda Insights enhanced monitoring on new and existing Lambda functions. When you enable Lambda Insights on a function in the Lambda console for a supported runtime, Lambda adds the Lambda Insights [extension](https://docs.aws.amazon.com/lambda/latest/dg/lambda-extended-runtimes.html) as a layer to your function, and verifies or attempts to attach the [CloudWatchLambdaInsightsExecutionRolePolicy](https://docs.aws.amazon.com/lambda/latest/dg/lambda-extended-runtimes.html) policy to your function's [execution role](https://docs.aws.amazon.com/lambda/latest/dg/lambda-extended-runtimes.html).

**To enable Lambda Insights in the Lambda console**

1. Open the [Functions page](https://docs.aws.amazon.com/lambda/latest/dg/lambda-extended-runtimes.html) of the Lambda console.
2. Choose your function.
6. Choose Save.

**Enabling Lambda Insights programmatically**

You can also enable Lambda Insights using the AWS Command Line Interface (AWS CLI), AWS Serverless Application Model (SAM) CLI, AWS CloudFormation, or the AWS Cloud Development Kit (AWS CDK). When you enable Lambda Insights programmatically on a function for a supported runtime, CloudWatch attaches the [CloudWatchLambdaInsightsExecutionRolePolicy](https://docs.aws.amazon.com/lambda/latest/dg/lambda-extended-runtimes.html) policy to your function's [execution role](https://docs.aws.amazon.com/lambda/latest/dg/lambda-extended-runtimes.html).
For more information, see [Getting started with Lambda Insights](https://docs.aws.amazon.com/lambda/latest/dg/lambda-insights-getting-started.html) in the *Amazon CloudWatch User Guide*.

**Using the Lambda Insights dashboard**

The Lambda Insights dashboard has two views in the CloudWatch console: the multi-function overview and the single-function view. The multi-function overview aggregates the runtime metrics for the Lambda functions in the current AWS account and Region. The single-function view shows the available runtime metrics for a single Lambda function.

You can use the Lambda Insights dashboard multi-function overview in the CloudWatch console to identify over- and under-utilized Lambda functions. You can use the Lambda Insights dashboard single-function view in the CloudWatch console to troubleshoot individual requests.

**To view the runtime metrics for all functions**

1. Open the [Multi-function](https://docs.aws.amazon.com/lambda/latest/dg/lambda-insights-getting-started.html) page in the CloudWatch console.
2. Choose from the predefined time ranges, or choose a custom time range.
3. (Optional) Choose **Add to dashboard** to add the widgets to your CloudWatch dashboard.
To view the runtime metrics of a single function

1. Open the Single-function page in the CloudWatch console.
2. Choose from the predefined time ranges, or choose a custom time range.
3. (Optional) Choose Add to dashboard to add the widgets to your CloudWatch dashboard.

For more information, see Creating and working with widgets on CloudWatch dashboards.

Example workflow to detect function anomalies

You can use the multi-function overview on the Lambda Insights dashboard to identify and detect compute memory anomalies with your function. For example, if the multi-function overview indicates that a function is using a large amount of memory, you can view detailed memory utilization metrics in the Memory Usage pane. You can then go to the Metrics dashboard to enable anomaly detection or create an alarm.

To enable anomaly detection for a function

1. Open the Multi-function page in the CloudWatch console.
2. Under Function summary, choose your function's name.
The single-function view opens with the function runtime metrics.

3. On the **Memory Usage** pane, choose the three vertical dots, and then choose **View in metrics** to open the **Metrics** dashboard.

4. On the **Graphed metrics** tab, in the **Actions** column, choose the first icon to enable anomaly detection for the function.
Example workflow using queries to troubleshoot a function

You can use the single-function view on the Lambda Insights dashboard to identify the root cause of a spike in function duration. For example, if the multi-function overview indicates a large increase in function duration, you can pause on or choose each function in the Duration pane to determine which function is causing the increase. You can then go to the single-function view and review the Application logs to determine the root cause.

To run queries on a function

1. Open the Multi-function page in the CloudWatch console.
2. In the Duration pane, choose your function to filter the duration metrics.
3. Open the Single-function page.
4. Choose the Filter metrics by function name dropdown list, and then choose your function.
5. To view the Most recent 1000 application logs, choose the Application logs tab.
6. Review the **Timestamp** and **Message** to identify the invocation request that you want to troubleshoot.

![Most recent 1000 application logs](image)

7. To show the **Most recent 1000 invocations**, choose the **Invocations** tab.

8. Select the **Timestamp** or **Message** for the invocation request that you want to troubleshoot.

![Most recent 1000 invocations](image)

9. Choose the **View logs** dropdown list, and then choose **View performance logs**.

   An autogenerated query for your function opens in the **Logs Insights** dashboard.

10. Choose **Run query** to generate a **Logs** message for the invocation request.
What's next?

- Learn how to create a CloudWatch Logs dashboard in [Create a Dashboard](#) in the Amazon CloudWatch User Guide.

- Learn how to add queries to a CloudWatch Logs dashboard in [Add Query to Dashboard or Export Query Results](#) in the Amazon CloudWatch User Guide.
Using CodeGuru Profiler with your Lambda function

You can use Amazon CodeGuru Profiler to gain insights into runtime performance of your Lambda functions. This page describes how to activate CodeGuru Profiler from the Lambda console.

Sections

- Supported runtimes
- Activating CodeGuru Profiler from the Lambda console
- What happens when you activate CodeGuru Profiler from the Lambda console?
- What's next?

Supported runtimes

You can activate CodeGuru Profiler from the Lambda console if your function's runtime is Python3.8, Python3.9, Java 8 with Amazon Linux 2, or Java 11. For additional runtime versions, you can activate CodeGuru Profiler manually.

- For Java runtimes, see Profiling your Java applications that run on AWS Lambda.
- For Python runtimes, see Profiling your Python applications that run on AWS Lambda.

Important

CodeGuru Profiler currently only supports functions that use x86_64 architecture.

Activating CodeGuru Profiler from the Lambda console

This section describes how to activate CodeGuru Profiler from the Lambda console.

To activate CodeGuru Profiler from the Lambda console

1. Open the Functions page of the Lambda console.
2. Choose your function.
3. Choose the Configuration tab.
4. On the Monitoring and operations tools pane, choose Edit.
6. Choose **Save**.

After activation, CodeGuru automatically creates a profiler group with the name `aws-lambda-<your-function-name>`. You can change the name from the CodeGuru console.

**What happens when you activate CodeGuru Profiler from the Lambda console?**

When you activate CodeGuru Profiler from the console, Lambda automatically does the following on your behalf:

- Lambda adds a CodeGuru Profiler layer to your function. For more details, see [Use AWS Lambda layers](#) in the *Amazon CodeGuru Profiler User Guide*.
- Lambda also adds environment variables to your function. The exact value varies based on the runtime.

### Environment variables

<table>
<thead>
<tr>
<th>Runtimes</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>java8.al2, java11</td>
<td>JAVA_TOOL_OPTIONS</td>
<td>-javaagent:/opt/codeguru-profiler-java-agent-standalone.jar</td>
</tr>
<tr>
<td>python3.8, python3.9</td>
<td>AWS_LAMBDA_EXEC_Wrapper</td>
<td>/opt/codeguru_profiler_lambda_exec</td>
</tr>
</tbody>
</table>

- Lambda adds the `AmazonCodeGuruProfilerAgentAccess` policy to your function's execution role.

---

**Note**

When you deactivate CodeGuru Profiler from the console, Lambda automatically removes the CodeGuru Profiler layer from your function. However, Lambda does not remove the environment variables or the `AmazonCodeGuruProfilerAgentAccess` policy from your execution role.
What's next?

- Learn more about the data collected by your profiler group in Working with visualizations in the Amazon CodeGuru Profiler User Guide.
**Example workflows using other AWS services**

AWS Lambda integrates with other AWS services to help you monitor, trace, debug, and troubleshoot your Lambda functions. This page shows workflows you can use with AWS X-Ray and AWS Trusted Advisor to trace and troubleshoot your Lambda functions.

**Sections**

- Prerequisites
- Pricing
- Example AWS X-Ray workflow to view a trace map
- Example AWS X-Ray workflow to view trace details
- Example AWS Trusted Advisor workflow to view recommendations
- What's next?

**Prerequisites**

The following section describes the steps to using AWS X-Ray and Trusted Advisor to troubleshoot your Lambda functions.

**Using AWS X-Ray**

AWS X-Ray needs to be enabled on the Lambda console to complete the AWS X-Ray workflows on this page. If your execution role does not have the required permissions, the Lambda console will attempt to add them to your execution role.

**To enable AWS X-Ray on the Lambda console**

1. Open the [Functions page](#) of the Lambda console.
2. Choose your function.
3. Choose the [Configuration](#) tab.
4. On the [Monitoring tools](#) pane, choose [Edit](#).
5. Under [AWS X-Ray](#), turn on [Active tracing](#).
6. Choose [Save](#).
Using AWS Trusted Advisor

AWS Trusted Advisor inspects your AWS environment and makes recommendations on ways you can save money, improve system availability and performance, and help close security gaps. You can use Trusted Advisor checks to evaluate the Lambda functions and applications in your AWS account. The checks provide recommended steps to take and resources for more information.

- For more information on AWS support plans for Trusted Advisor checks, see Support plans.
- For more information about the checks for Lambda, see AWS Trusted Advisor best practice checklist.
- For more information on how to use the Trusted Advisor console, see Get started with AWS Trusted Advisor.
- For instructions on how to allow and deny console access to Trusted Advisor, see IAM policy examples.

Pricing

- With AWS X-Ray you pay only for what you use, based on the number of traces recorded, retrieved, and scanned. For more information, see AWS X-Ray Pricing.
- Trusted Advisor cost optimization checks are included with AWS Business and Enterprise support subscriptions. For more information, see AWS Trusted Advisor Pricing.

Example AWS X-Ray workflow to view a trace map

If you've enabled AWS X-Ray, you can view a trace map on the CloudWatch console. A trace map displays your service endpoints and resources as nodes and highlights the traffic, latency, and errors for each node and its connections.

You can choose a node to see detailed insights about the correlated metrics, logs, and traces associated with that part of the service. This enables you to investigate problems and their effect on an application.

To view trace map and traces using the CloudWatch console

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose Monitoring.
5. Choose Trace Map under X-Ray traces from the left navigation pane.
6. Choose from the predefined time ranges, or choose a custom time range.
7. To troubleshoot requests, choose a filter.

**Example AWS X-Ray workflow to view trace details**

If you've enabled AWS X-Ray, you can use the single-function view on the CloudWatch Lambda Insights dashboard to show the distributed trace data of a function invocation error. For example, if the application logs message shows an error, you can open the trace map to see the distributed trace data and the other services handling the transaction.

**To view trace details of a function**

1. Open the single-function view in the CloudWatch console.
2. Choose the Application logs tab.
3. Use the Timestamp or Message to identify the invocation request that you want to troubleshoot.
4. To show the Most recent 1000 invocations, choose the Invocations tab.

   ![Invocations tab](image-url)

5. Choose the Request ID column to sort entries in ascending alphabetical order.
6. In the Trace column, choose View.

   The Trace details page opens in the trace map view.
Example AWS Trusted Advisor workflow to view recommendations

Trusted Advisor checks Lambda functions in all AWS Regions to identify functions with the highest potential cost savings, and deliver actionable recommendations for optimization. It analyzes your Lambda usage data such as function execution time, billed duration, memory used, memory configured, timeout configuration and errors.

For example, the *Lambda Functions with High Error Rate* check recommends that you use AWS X-Ray or CloudWatch to detect errors with your Lambda functions.

**To check for functions with high error rates**

1. Open the [Trusted Advisor](#) console.
2. Choose the **Cost Optimization** category.
3. Scroll down to *AWS Lambda Functions with High Error Rates*. Expand the section to see the results and the recommended actions.

**What's next?**

- Learn more about how to integrate traces, metrics, logs, and alarms in [Using the X-Ray trace map](#).
- Learn more about how to get a list of Trusted Advisor checks in [Using Trusted Advisor as a web service](#).
Working with Lambda layers

A Lambda layer is a .zip file archive that contains supplementary code or data. Layers usually contain library dependencies, a custom runtime, or configuration files.

There are multiple reasons why you might consider using layers:

- **To reduce the size of your deployment packages.** Instead of including all of your function dependencies along with your function code in your deployment package, put them in a layer. This keeps deployment packages small and organized.

- **To separate core function logic from dependencies.** With layers, you can update your function dependencies independent of your function code, and vice versa. This promotes separation of concerns and helps you focus on your function logic.

- **To share dependencies across multiple functions.** After you create a layer, you can apply it to any number of functions in your account. Without layers, you need to include the same dependencies in each individual deployment package.

- **To use the Lambda console code editor.** The code editor is a useful tool for testing minor function code updates quickly. However, you can’t use the editor if your deployment package size is too large. Using layers reduces your package size and can unlock usage of the code editor.

The following diagram illustrates the high-level architectural differences between two functions that share dependencies. One uses Lambda layers, and the other does not.
When you add a layer to a function, Lambda extracts the layer contents into the /opt directory in your function’s execution environment. All natively supported Lambda runtimes include paths to specific directories within the /opt directory. This gives your function access to your layer content. For more information about these specific paths and how to properly package your layers, see the section called “Packaging layers”.

You can include up to five layers per function. Also, you can use layers only with Lambda functions deployed as a .zip file archive. For functions defined as a container image, package your preferred
runtime and all code dependencies when you create the container image. For more information, see [Working with Lambda layers and extensions in container images](https://aws.amazon.com/compute/blog/working-with-lambda-layers-and-extensions-in-container-images) on the AWS Compute Blog.

**Topics**

- How to use layers
- Layers and layer versions
- Sample applications
- Packaging your layer content
- Creating and deleting layers in Lambda
- Adding layers to functions
- Using AWS CloudFormation with layers
- Using AWS SAM with layers

**How to use layers**

To create a layer, package your dependencies into a .zip file, similar to how you create a normal deployment package. More specifically, the general process of creating and using layers involves these three steps:

- **First, package your layer content.** This means creating a .zip file archive. For more information, see the section called “Packaging layers”.
- **Next, create the layer in Lambda.** For more information, see the section called “Creating and deleting layers”.
- **Add the layer to your function(s).** For more information, see the section called “Adding layers”.

**Layers and layer versions**

A layer version is an immutable snapshot of a specific version of a layer. When you create a new layer, Lambda creates a new layer version with a version number of 1. Each time you publish an update to the layer, Lambda increments the version number and creates a new layer version.

Every layer version is identified by a unique Amazon Resource Name (ARN). When adding a layer to the function, you must specify the exact layer version you want to use.
Sample applications

The GitHub repository for this guide provides blank sample applications that demonstrate the use of layers for dependency management.

- **Node.js** – [blank-nodejs](#)
- **Python** – [blank-python](#)
- **Java** – [blank-java](#)
- **Ruby** – [blank-ruby](#)
Packaging your layer content

A Lambda layer is a .zip file archive that contains supplementary code or data. Layers usually contain library dependencies, a custom runtime, or configuration files.

This section explains how to properly package your layer content. For more conceptual information about layers and why you might consider using them, see Lambda layers.

The first step to creating a layer is to bundle all of your layer content into a .zip file archive. Because Lambda functions run on Amazon Linux, your layer content must be able to compile and build in a Linux environment. If you build packages on your local Windows or Mac machine, you’ll get output binaries for that operating system by default. These binaries may not work properly when you upload them to Lambda.

To ensure that your layer content works properly in a Linux environment, we recommend creating your layer content using AWS Cloud9. AWS Cloud9 is a cloud-based integrated development environment (IDE) that provides built-in access to a Linux server for running and testing code. For more information, see Using Lambda layers to simplify your development process on the AWS Compute Blog.

Topics

• Layer paths for each Lambda runtime

Layer paths for each Lambda runtime

When you add a layer to a function, Lambda loads the layer content into the /opt directory of that execution environment. For each Lambda runtime, the PATH variable already includes specific folder paths within the /opt directory. To ensure that your layer content gets picked up by the PATH variable, include the content in the following folder paths:

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node.js</td>
<td>nodejs/node_modules</td>
</tr>
<tr>
<td></td>
<td>nodejs/node14/node_modules (NODE_PATH)</td>
</tr>
<tr>
<td></td>
<td>nodejs/node16/node_modules (NODE_PATH)</td>
</tr>
</tbody>
</table>
The following examples show how you can structure the folders in your layer .zip archive.

Node.js

Example file structure for the AWS X-Ray SDK for Node.js

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

Python

Example file structure for the Pillow library

```
pillow.zip
# python/PIL
# python/Pillow-5.3.0.dist-info
```

Ruby

Example file structure for the JSON gem

```
json.zip
```
Java

**Example file structure for the Jackson JAR file**

```plaintext
jackson.zip
# java/lib/jackson-core-2.2.3.jar
```

All

**Example file structure for the jq library**

```plaintext
jq.zip
# bin/jq
```
Creating and deleting layers in Lambda

A Lambda layer is a .zip file archive that contains supplementary code or data. Layers usually contain library dependencies, a custom runtime, or configuration files.

This section explains how to create and delete layers in Lambda. For more conceptual information about layers and why you might consider using them, see Lambda layers.

After you’ve packaged your layer content, the next step is to create the layer in Lambda. This section demonstrates how to create and delete layers using the Lambda console or the Lambda API only. To create a layer using AWS CloudFormation, see the section called “Layers with AWS CloudFormation”. To create a layer using the AWS Serverless Application Model (AWS SAM), see the section called “Layers with AWS SAM”.

Topics

- Creating a layer
- Deleting a layer version

Creating a layer

To create a layer, you can either upload the .zip file archive from your local machine or from Amazon Simple Storage Service (Amazon S3). Lambda extracts the layer contents into the /opt directory when setting up the execution environment for the function.

Layers can have one or more layer versions. When you create a layer, Lambda sets the layer version to version 1. You can change the permissions on an existing layer version at any time. However, to update the code or make other configuration changes, you must create a new version of the layer.

To create a layer (console)

1. Open the Layers page of the Lambda console.
2. Choose Create layer.
3. Under Layer configuration, for Name, enter a name for your layer.
4. (Optional) For Description, enter a description for your layer.
5. To upload your layer code, do one of the following:
   - To upload a .zip file from your computer, choose Upload a .zip file. Then, choose Upload to select your local .zip file.
• To upload a file from Amazon S3, choose **Upload a file from Amazon S3**. Then, for **Amazon S3 link URL**, enter a link to the file.

6. (Optional) For **Compatible architectures**, choose one value or both values. For more information, see [the section called “Instruction sets (ARM/x86)”](#).

7. (Optional) For **Compatible runtimes**, choose the runtimes that your layer is compatible with.

8. (Optional) For **License**, enter any necessary license information.

9. Choose **Create**.

Alternatively, you can also use the [PublishLayerVersion](#) API to create a layer. For example, you can use the publish-layer-version AWS Command Line Interface (CLI) command with a name, description, and .zip file archive specified. The license info, compatible runtimes, and compatible architecture parameters are optional.

```bash
aws lambda publish-layer-version --layer-name my-layer \
  --description "My layer" \
  --license-info "MIT" \
  --zip-file fileb://layer.zip \
  --compatible-runtimes python3.10 python3.11 \
  --compatible-architectures "arm64" "x86_64"
```

You should see output similar to the following:

```json
{
    "Content": {
        "Location": "https://awslambda-us-east-2-layers.s3.us-east-2.amazonaws.com/
        snapshots/123456789012/my-layer-4aaa2fbb-ff77-4b0a-ad92-5b78a716a96a?
        versionId=27iWyA73cCAYqyH...",
        "CodeSha256": "tv9jJ0+rPbXUUXuRKi7CwHzKtLDkDRJLB3cC3Z/ouXo=",
        "CodeSize": 169
    },
    "Description": "My layer",
    "CreatedDate": "2023-11-14T23:03:52.894+0000",
    "Version": 1,
    "CompatibleArchitectures": [
        "arm64",
        "x86_64"
    ],
}
```
Each time you call `publish-layer-version`, you create a new version of the layer.

## Deleting a layer version

To delete a layer version, use the [DeleteLayerVersion](https://docs.aws.amazon.com/lambda/latest/dg/API_DeleteLayerVersion.html) API. For example, you can use the `delete-layer-version` CLI command with the layer name and layer version specified.

```bash
aws lambda delete-layer-version --layer-name my-layer --version-number 1
```

When you delete a layer version, you can no longer configure a Lambda function to use it. However, any function that already uses the version continues to have access to it. Also, Lambda never reuses version numbers for a layer name.
Adding layers to functions

A Lambda layer is a .zip file archive that contains supplementary code or data. Layers usually contain library dependencies, a custom runtime, or configuration files.

This section explains how to add a layer to a Lambda function. For more conceptual information about layers and why you might consider using them, see Lambda layers.

Before you can configure a Lambda function to use a layer, you must:

- Package your layer content
- Create a layer in Lambda
- Make sure that you have permission to call the GetLayerVersion API on the layer version. For functions in your AWS account, you must have this permission in your user policy. To use a layer in another account, the owner of that account must grant your account permission in a resource-based policy. For examples, see the section called “Granting layer access to other accounts”.

You can add up to five layers to a Lambda function. The total unzipped size of the function and all layers cannot exceed the unzipped deployment package size quota of 250 MB. For more information, see Lambda quotas.

Your functions can continue to use any layer version that you've already added, even after that layer version has been deleted, or after your permission to access the layer is revoked. However, you cannot create a new function that uses a deleted layer version.

Note

Make sure that the layers you add to a function are compatible with the runtime and instruction set architecture of the function.

To add a layer to a function (console)

1. Open the Functions page of the Lambda console.
2. Choose the function to configure.
3. Under Layers, choose Add a layer
4. Under Choose a layer, choose a layer source:
a. For the **AWS layers** or **Custom layers** layer sources, choose a layer from the pull-down menu. Under **Version**, choose a layer version from the pull-down menu.

b. For the **Specify an ARN** layer source, enter an ARN in the text box and choose **Verify**. Then, choose **Add**.

The order in which you add the layers is the order in which Lambda merges the layer content into the execution environment. You can change the layer merge order using the console.

**To update layer merge order for your function (console)**

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to configure.
3. Under **Layers**, choose **Edit**
4. Choose one of the layers.
5. Choose **Merge earlier** or **Merge later** to adjust the order of the layers.
6. Choose **Save**.

Layers are versioned. The content of each layer version is immutable. The owner of a layer can release new layer versions to provide updated content. You can use the console to update the layer version attached to your functions.

**To update layer versions for your function (console)**

1. Open the [Layers page](#) of the Lambda console.
2. Choose the layer you want to update the version for.
3. Choose the **Functions using this version** tab.
4. Choose the functions you want to modify, then choose **Edit**.
5. For **Layer version**, choose the layer version to change to.
6. Choose **Update functions**.

You cannot update function layer versions across AWS accounts.

**Topics**

- [Accessing layer content from your function](#)
Accessing layer content from your function

If your Lambda function includes layers, Lambda extracts the layer contents into the /opt directory in the function execution environment. Lambda extracts the layers in the order (low to high) listed by the function. Lambda merges folders with the same name. If the same file appears in multiple layers, the function uses the version in the last extracted layer.

Each Lambda runtime adds specific /opt directory folders to the PATH variable. Your function code can access the layer content without having to specify the path. For more information about path settings in the Lambda execution environment, see the section called “Defined runtime environment variables”.

Refer to the section called “Layer paths for each Lambda runtime” to learn where to include your libraries when creating a layer.

If you’re using a Node.js or Python runtime, you can use the built-in code editor in the Lambda console. You should be able to import any library that you’ve added as a layer to the current function.

Finding layer information

To find layers in your account that are compatible with your function’s runtime, use the ListLayers API. For example, you can use the following list-layers AWS Command Line Interface (CLI) command:

```
aws lambda list-layers --compatible-runtime python3.9
```

You should see output similar to the following:

```
{
   "Layers": [
   {
      "LayerName": "my-layer",
      "LatestMatchingVersion": {
      
      
```
To list all layers in your account, omit the `--compatible-runtime` option. The response details show the latest version of each layer.

You can also get the latest version of a layer using the [ListLayerVersions](https://docs.aws.amazon.com/lambda/latest/dg/API_ListLayerVersions.html) API. For example, you can use the following `list-layer-versions` CLI command:

```bash
aws lambda list-layer-versions --layer-name my-layer
```

You should see output similar to the following:

```json
{
    "LayerVersions": [
        {
            "Version": 2,
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:37:46.592+0000",
            "CompatibleRuntimes": ["java11" ]
        },
        {
            "Version": 1,
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:27:46.592+0000",
            "CompatibleRuntimes": ["python3.9", "python3.10", "python3.11"]
        }
    ]
}
```
"java11"
]
}
]}

Finding layer information
Using AWS CloudFormation with layers

You can use AWS CloudFormation to create a layer and associate the layer with your Lambda function. The following example template creates a layer named my-lambda-layer and attaches the layer to the Lambda function using the Layers property.

---

Description: CloudFormation Template for Lambda Function with Lambda Layer

Resources:

MyLambdaLayer:
  Type: AWS::Lambda::LayerVersion
  Properties:
    LayerName: my-lambda-layer
    Description: My Lambda Layer
    Content:
      S3Bucket: my-bucket
      S3Key: my-layer.zip
    CompatibleRuntimes:
      - python3.9
      - python3.10
      - python3.11

MyLambdaFunction:
  Type: AWS::Lambda::Function
  Properties:
    FunctionName: my-lambda-function
    Runtime: python3.9
    Handler: index.handler
    Timeout: 10
    Policies:
      - AWSLambdaBasicExecutionRole
      - AWSLambda_ReadOnlyAccess
      - AWSXrayWriteOnlyAccess
    Layers:
      - !Ref MyLambdaLayer
Using AWS SAM with layers

You can use the AWS Serverless Application Model (AWS SAM) to automate the creation of layers in your application. The `AWS::Serverless::LayerVersion` resource type creates a layer version that you can reference from your Lambda function configuration.

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: AWS SAM Template for Lambda Function with Lambda Layer

Resources:
  MyLambdaLayer:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      ContentUri: s3://my-bucket/my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: MyLambdaFunction
      Runtime: python3.9
      Handler: app.handler
      CodeUri: s3://my-bucket/my-function
      Layers:
        - !Ref MyLambdaLayer
```
Lambda extensions

You can use Lambda extensions to augment your Lambda functions. For example, use Lambda extensions to integrate functions with your preferred monitoring, observability, security, and governance tools. You can choose from a broad set of tools that AWS Lambda Partners provides, or you can create your own Lambda extensions.

Lambda supports external and internal extensions. An external extension runs as an independent process in the execution environment and continues to run after the function invocation is fully processed. Because extensions run as separate processes, you can write them in a different language than the function. All Lambda runtimes support extensions.

An internal extension runs as part of the runtime process. Your function accesses internal extensions by using wrapper scripts or in-process mechanisms such as JAVA_TOOL_OPTIONS. For more information, see Modifying the runtime environment.

You can add extensions to a function using the Lambda console, the AWS Command Line Interface (AWS CLI), or infrastructure as code (IaC) services and tools such as AWS CloudFormation, AWS Serverless Application Model (AWS SAM), and Terraform.

You are charged for the execution time that the extension consumes (in 1 ms increments). There is no cost to install your own extensions. For more pricing information for extensions, see AWS Lambda Pricing. For pricing information for partner extensions, see those partners' websites. See the section called “Extensions partners” for a list of official partner extensions.

For a tutorial on extensions and how to use them with your Lambda functions, see the AWS Lambda Extensions Workshop.

Topics

• Execution environment
• Impact on performance and resources
• Permissions
• Configuring Lambda extensions
• AWS Lambda extensions partners
• Lambda Extensions API
• Lambda Telemetry API
Execution environment

Lambda invokes your function in an execution environment, which provides a secure and isolated runtime environment. The execution environment manages the resources required to run your function and provides lifecycle support for the function’s runtime and extensions.

The lifecycle of the execution environment includes the following phases:

- **Init**: In this phase, Lambda creates or unfreezes an execution environment with the configured resources, downloads the code for the function and all layers, initializes any extensions, initializes the runtime, and then runs the function's initialization code (the code outside the main handler). The Init phase happens either during the first invocation, or in advance of function invocations if you have enabled provisioned concurrency.

  The Init phase is split into three sub-phases: Extension init, Runtime init, and Function init. These sub-phases ensure that all extensions and the runtime complete their setup tasks before the function code runs.

  When Lambda SnapStart is activated, the Init phase happens when you publish a function version. Lambda saves a snapshot of the memory and disk state of the initialized execution environment, persists the encrypted snapshot, and caches it for low-latency access. If you have a beforeCheckpoint runtime hook, then the code runs at the end of Init phase.

- **Restore** (SnapStart only): When you first invoke a SnapStart function and as the function scales up, Lambda resumes new execution environments from the persisted snapshot instead of initializing the function from scratch. If you have an afterRestore() runtime hook, the code runs at the end of the Restore phase. You are charged for the duration of afterRestore() runtime hooks. The runtime (JVM) must load and afterRestore() runtime hooks must complete within the timeout limit (10 seconds). Otherwise, you'll get a SnapStartTimeoutException. When the Restore phase completes, Lambda invokes the function handler (the Invoke phase).

- **Invoke**: In this phase, Lambda invokes the function handler. After the function runs to completion, Lambda prepares to handle another function invocation.

- **Shutdown**: This phase is triggered if the Lambda function does not receive any invocations for a period of time. In the Shutdown phase, Lambda shuts down the runtime, alerts the extensions to let them stop cleanly, and then removes the environment. Lambda sends a Shutdown event to each extension, which tells the extension that the environment is about to be shut down.
During the Init phase, Lambda extracts layers containing extensions into the /opt directory in the execution environment. Lambda looks for extensions in the /opt/extensions/ directory, interprets each file as an executable bootstrap for launching the extension, and starts all extensions in parallel.

**Impact on performance and resources**

The size of your function's extensions counts towards the deployment package size limit. For a .zip file archive, the total unzipped size of the function and all extensions cannot exceed the unzipped deployment package size limit of 250 MB.

Extensions can impact the performance of your function because they share function resources such as CPU, memory, and storage. For example, if an extension performs compute-intensive operations, you may see your function's execution duration increase.

Each extension must complete its initialization before Lambda invokes the function. Therefore, an extension that consumes significant initialization time can increase the latency of the function invocation.

To measure the extra time that the extension takes after the function execution, you can use the PostRuntimeExtensionsDuration function metric. To measure the increase in memory used, you can use the MaxMemoryUsed metric. To understand the impact of a specific extension, you can run different versions of your functions side by side.

**Permissions**

Extensions have access to the same resources as functions. Because extensions are executed within the same environment as the function, permissions are shared between the function and the extension.

For a .zip file archive, you can create an AWS CloudFormation template to simplify the task of attaching the same extension configuration—including AWS Identity and Access Management (IAM) permissions—to multiple functions.
Configuring Lambda extensions

Configuring extensions (.zip file archive)

You can add an extension to your function as a Lambda layer. Using layers enables you to share extensions across your organization or to the entire community of Lambda developers. You can add one or more extensions to a layer. You can register up to 10 extensions for a function.

You add the extension to your function using the same method as you would for any layer. For more information, see Lambda layers.

Add an extension to your function (console)

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose the Code tab if it is not already selected.
5. For Choose a layer, choose Specify an ARN.
6. For Specify an ARN, enter the Amazon Resource Name (ARN) of an extension layer.
7. Choose Add.

Using extensions in container images

You can add extensions to your container image. The ENTRYPOINT container image setting specifies the main process for the function. Configure the ENTRYPOINT setting in the Dockerfile, or as an override in the function configuration.

You can run multiple processes within a container. Lambda manages the lifecycle of the main process and any additional processes. Lambda uses the Extensions API to manage the extension lifecycle.

Example: Adding an external extension

An external extension runs in a separate process from the Lambda function. Lambda starts a process for each extension in the /opt/extensions/ directory. Lambda uses the Extensions API to manage the extension lifecycle. After the function has run to completion, Lambda sends a Shutdown event to each external extension.
Example of adding an external extension to a Python base image

```dockerfile
FROM public.ecr.aws/lambda/python:3.11

# Copy and install the app
COPY /app /app
WORKDIR /app
RUN pip install -r requirements.txt

# Add an extension from the local directory into /opt
ADD my-extension.zip /opt
CMD python ./my-function.py
```

Next steps

To learn more about extensions, we recommend the following resources:

- For a basic working example, see [Building Extensions for AWS Lambda](https://aws.amazon.com/blogs/compute/building-extensions-for-aws-lambda) on the AWS Compute Blog.
- For information about extensions that AWS Lambda Partners provides, see [Introducing AWS Lambda Extensions](https://aws.amazon.com/blogs/compute/introducing-aws-lambda-extensions) on the AWS Compute Blog.
- To view available example extensions and wrapper scripts, see [AWS Lambda Extensions](https://github.com/aws-samples/aws-lambda-/extensions) on the AWS Samples GitHub repository.
AWS Lambda extensions partners

AWS Lambda has partnered with several third party entities to provide extensions to integrate with your Lambda functions. The following list details third party extensions that are ready for you to use at any time.

- **AppDynamics** – Provides automatic instrumentation of Node.js or Python Lambda functions, providing visibility and alerting on function performance.
- **Check Point CloudGuard** – An extension-based runtime solution that offers full lifecycle security for serverless applications.
- **Datadog** – Provides comprehensive, real-time visibility to your serverless applications through the use of metrics, traces, and logs.
- **Dynatrace** – Provides visibility into traces and metrics, and leverages AI for automated error detection and root cause analysis across the entire application stack.
- **Elastic** – Provides Application Performance Monitoring (APM) to identify and resolve root cause issues using correlated traces, metrics, and logs.
- **Epsagon** – Listens to invocation events, stores traces, and sends them in parallel to Lambda function executions.
- **Fastly** – Protects your Lambda functions from suspicious activity, such as injection-style attacks, account takeover via credential stuffing, malicious bots, and API abuse.
- **HashiCorp Vault** – Manages secrets and makes them available for developers to use within function code, without making functions Vault aware.
- **Honeycomb** – Observability tool for debugging your app stack.
- **Lumigo** – Profiles Lambda function invocations and collects metrics for troubleshooting issues in serverless and microservice environments.
- **New Relic** – Runs alongside Lambda functions, automatically collecting, enhancing, and transporting telemetry to New Relic's unified observability platform.
- **Sedai** – An autonomous cloud management platform, powered by AI/ML, that delivers continuous optimization for cloud operations teams to maximize cloud cost savings, performance, and availability at scale.
- **Sentry** – Diagnose, fix, and optimize performance of Lambda functions.
- **Site24x7** – Achieve real-time observability into your Lambda environments
- **Splunk** – Collects high-resolution, low-latency metrics for efficient and effective monitoring of Lambda functions.
• **Sumo Logic** – Provides visibility into the health and performance of serverless applications.

• **Thundra** – Provides asynchronous telemetry reporting, such as traces, metrics, and logs.

**AWS managed extensions**

AWS provides its own managed extensions, including:

• **AWS AppConfig** – Use feature flags and dynamic data to update your Lambda functions. You can also use this extension to update other dynamic configuration, such as ops throttling and tuning.

• **Amazon CodeGuru Profiler** – Improves application performance and reduces cost by pinpointing an application's most expensive line of code and providing recommendations for improving code.

• **CloudWatch Lambda Insights** – Monitor, troubleshoot, and optimize the performance of your Lambda functions through automated dashboards.

• **AWS Distro for OpenTelemetry (ADOT)** – Enables functions to send trace data to AWS monitoring services such as AWS X-Ray, and to destinations that support OpenTelemetry such as Honeycomb and Lightstep.

• **AWS Parameters and Secrets** – Enables customers to securely retrieve parameters from **AWS Systems Manager Parameter Store** and secrets from **AWS Secrets Manager**.

For additional extensions samples and demo projects, see [AWS Lambda Extensions](#).
Lambda Extensions API

Lambda function authors use extensions to integrate Lambda with their preferred tools for monitoring, observability, security, and governance. Function authors can use extensions from AWS, AWS Partners, and open-source projects. For more information on using extensions, see Introducing AWS Lambda Extensions on the AWS Compute Blog. This section describes how to use the Lambda Extensions API, the Lambda execution environment lifecycle, and the Lambda Extensions API reference.

As an extension author, you can use the Lambda Extensions API to integrate deeply into the Lambda execution environment. Your extension can register for function and execution environment lifecycle events. In response to these events, you can start new processes, run logic, and control and participate in all phases of the Lambda lifecycle: initialization, invocation, and shutdown. In addition, you can use the Runtime Logs API to receive a stream of logs.

An extension runs as an independent process in the execution environment and can continue to run after the function invocation is fully processed. Because extensions run as processes, you can write them in a different language than the function. We recommend that you implement extensions using a compiled language. In this case, the extension is a self-contained binary that is compatible with supported runtimes. All Lambda runtimes support extensions. If you use a non-compiled language, ensure that you include a compatible runtime in the extension.

Lambda also supports internal extensions. An internal extension runs as a separate thread in the runtime process. The runtime starts and stops the internal extension. An alternative way to integrate with the Lambda environment is to use language-specific environment variables and wrapper scripts. You can use these to configure the runtime environment and modify the startup behavior of the runtime process.
You can add extensions to a function in two ways. For a function deployed as a .zip file archive, you deploy your extension as a layer. For a function defined as a container image, you add the extensions to your container image.

Note
For example extensions and wrapper scripts, see AWS Lambda Extensions on the AWS Samples GitHub repository.

Topics
- Lambda execution environment lifecycle
- Extensions API reference

Lambda execution environment lifecycle

The lifecycle of the execution environment includes the following phases:

- **Init**: In this phase, Lambda creates or unfreezes an execution environment with the configured resources, downloads the code for the function and all layers, initializes any extensions, initializes the runtime, and then runs the function's initialization code (the code outside the main handler). The Init phase happens either during the first invocation, or in advance of function invocations if you have enabled provisioned concurrency.

  The Init phase is split into three sub-phases: Extension init, Runtime init, and Function init. These sub-phases ensure that all extensions and the runtime complete their setup tasks before the function code runs.

- **Invoke**: In this phase, Lambda invokes the function handler. After the function runs to completion, Lambda prepares to handle another function invocation.

- **Shutdown**: This phase is triggered if the Lambda function does not receive any invocations for a period of time. In the Shutdown phase, Lambda shuts down the runtime, alerts the extensions to let them stop cleanly, and then removes the environment. Lambda sends a Shutdown event to each extension, which tells the extension that the environment is about to be shut down.
Each phase starts with an event from Lambda to the runtime and to all registered extensions. The runtime and each extension signal completion by sending a Next API request. Lambda freezes the execution environment when each process has completed and there are no pending events.

Topics

- Init phase
- Invoke phase
- Shutdown phase
- Permissions and configuration
- Failure handling
- Troubleshooting extensions
Init phase

During the Extension init phase, each extension needs to register with Lambda to receive events. Lambda uses the full file name of the extension to validate that the extension has completed the bootstrap sequence. Therefore, each Register API call must include the Lambda-Extension-Name header with the full file name of the extension.

You can register up to 10 extensions for a function. This limit is enforced through the Register API call.

After each extension registers, Lambda starts the Runtime init phase. The runtime process calls functionInit to start the Function init phase.

The Init phase completes after the runtime and each registered extension indicate completion by sending a Next API request.

Note

Extensions can complete their initialization at any point in the Init phase.
Invoke phase

When a Lambda function is invoked in response to a Next API request, Lambda sends an Invoke event to the runtime and to each extension that is registered for the Invoke event.

During the invocation, external extensions run in parallel with the function. They also continue running after the function has completed. This enables you to capture diagnostic information or to send logs, metrics, and traces to a location of your choice.

After receiving the function response from the runtime, Lambda returns the response to the client, even if extensions are still running.

The Invoke phase ends after the runtime and all extensions signal that they are done by sending a Next API request.
**Event payload:** The event sent to the runtime (and the Lambda function) carries the entire request, headers (such as RequestId), and payload. The event sent to each extension contains metadata that describes the event content. This lifecycle event includes the type of the event, the time that the function times out (deadlineMs), the requestId, the invoked function's Amazon Resource Name (ARN), and tracing headers.

Extensions that want to access the function event body can use an in-runtime SDK that communicates with the extension. Function developers use the in-runtime SDK to send the payload to the extension when the function is invoked.

Here is an example payload:

```json
{
    "eventType": "INVOKE",
    "deadlineMs": 676051,
    "requestId": "3da1f2dc-3222-475e-9205-e2e6c6318895",
    "invokedFunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:ExtensionTest",
    "tracing": {
        "type": "X-Amzn-Trace-Id",
        "value": "Root=1-5f35ae12-0c0fec141ab77a00bc047aa2;Parent=2be948a625588e32;Sampled=1"
    }
}
```
**Duration limit:** The function's timeout setting limits the duration of the entire Invoke phase. For example, if you set the function timeout as 360 seconds, the function and all extensions need to complete within 360 seconds. Note that there is no independent post-invoke phase. The duration is the sum of all invocation time (runtime + extensions) and is not calculated until the function and all extensions have finished running.

**Performance impact and extension overhead:** Extensions can impact function performance. As an extension author, you have control over the performance impact of your extension. For example, if your extension performs compute-intensive operations, the function's duration increases because the extension and the function code share the same CPU resources. In addition, if your extension performs extensive operations after the function invocation completes, the function duration increases because the Invoke phase continues until all extensions signal that they are completed.

**Note**

Lambda allocates CPU power in proportion to the function's memory setting. You might see increased execution and initialization duration at lower memory settings because the function and extension processes are competing for the same CPU resources. To reduce the execution and initialization duration, try increasing the memory setting.

To help identify the performance impact introduced by extensions on the Invoke phase, Lambda outputs the `PostRuntimeExtensionsDuration` metric. This metric measures the cumulative time spent between the runtime `Next` API request and the last extension `Next` API request. To measure the increase in memory used, use the `MaxMemoryUsed` metric. For more information about function metrics, see [Working with Lambda function metrics](#).

Function developers can run different versions of their functions side by side to understand the impact of a specific extension. We recommend that extension authors publish expected resource consumption to make it easier for function developers to choose a suitable extension.

**Shutdown phase**

When Lambda is about to shut down the runtime, it sends a `Shutdown` to each registered external extension. Extensions can use this time for final cleanup tasks. The `Shutdown` event is sent in response to a `Next` API request.

**Duration limit:** The maximum duration of the Shutdown phase depends on the configuration of registered extensions:
• 0 ms – A function with no registered extensions
• 500 ms – A function with a registered internal extension
• 2,000 ms – A function with one or more registered external extensions

For a function with external extensions, Lambda reserves up to 300 ms (500 ms for a runtime with an internal extension) for the runtime process to perform a graceful shutdown. Lambda allocates the remainder of the 2,000 ms limit for external extensions to shut down.

If the runtime or an extension does not respond to the Shutdown event within the limit, Lambda ends the process using a SIGKILL signal.

Event payload: The Shutdown event contains the reason for the shutdown and the time remaining in milliseconds.

The shutdownReason includes the following values:

• SPINDOWN – Normal shutdown
• TIMEOUT – Duration limit timed out
• FAILURE – Error condition, such as an out-of-memory event

```json
{
   "eventType": "SHUTDOWN",
   "shutdownReason": "reason for shutdown",
   "deadlineMs": "the time and date that the function times out in Unix time milliseconds"
}
```

Permissions and configuration

Extensions run in the same execution environment as the Lambda function. Extensions also share resources with the function, such as CPU, memory, and /tmp disk storage. In addition, extensions use the same AWS Identity and Access Management (IAM) role and security context as the function.

File system and network access permissions: Extensions run in the same file system and network name namespace as the function runtime. This means that extensions need to be compatible with the associated operating system. If an extension requires any additional outbound network traffic rules, you must apply these rules to the function configuration.

⚠️ Note
Because the function code directory is read-only, extensions cannot modify the function code.

Environment variables: Extensions can access the function's environment variables, except for the following variables that are specific to the runtime process:

• AWS_EXECUTION_ENV
• AWS_LAMBDA_LOG_GROUP_NAME
• AWS_LAMBDA_LOG_STREAM_NAME
• AWS_XRAY_CONTEXT_MISSING
• AWS_XRAY_DAEMON_ADDRESS
• LAMBDA_RUNTIME_DIR
• LAMBDA_TASK_ROOT
• _AWS_XRAY_DAEMON_ADDRESS
• _AWS_XRAY_DAEMON_PORT
• _HANDLER

Failure handling

Initialization failures: If an extension fails, Lambda restarts the execution environment to enforce consistent behavior and to encourage fail fast for extensions. Also, for some customers, the extensions must meet mission-critical needs such as logging, security, governance, and telemetry collection.

Invoke failures (such as out of memory, function timeout): Because extensions share resources with the runtime, memory exhaustion affects them. When the runtime fails, all extensions and the runtime itself participate in the Shutdown phase. In addition, the runtime is restarted—either automatically as part of the current invocation, or via a deferred re-initialization mechanism.

If there is a failure (such as a function timeout or runtime error) during Invoke, the Lambda service performs a reset. The reset behaves like a Shutdown event. First, Lambda shuts down the runtime, then it sends a Shutdown event to each registered external extension. The event includes the reason for the shutdown. If this environment is used for a new invocation, the extension and runtime are re-initialized as part of the next invocation.

For a more detailed explanation of the previous diagram, see Failures during the invoke phase.

Extension logs: Lambda sends the log output of extensions to CloudWatch Logs. Lambda also generates an additional log event for each extension during Init. The log event records the name and registration preference (event, config) on success, or the failure reason on failure.

Troubleshooting extensions

• If a Register request fails, make sure that the Lambda-Extension-Name header in the Register API call contains the full file name of the extension.
If the Register request fails for an internal extension, make sure that the request does not register for the Shutdown event.

**Extensions API reference**

The OpenAPI specification for the extensions API version 2020-01-01 is available here: [extensions-api.zip](extensions-api.zip)

You can retrieve the value of the API endpoint from the AWS_LAMBDA_RUNTIME_API environment variable. To send a Register request, use the prefix 2020-01-01/ before each API path. For example:

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
```

**API methods**

- Register
- Next
- Init error
- Exit error

**Register**

During Extension init, all extensions need to register with Lambda to receive events. Lambda uses the full file name of the extension to validate that the extension has completed the bootstrap sequence. Therefore, each Register API call must include the Lambda-Extension-Name header with the full file name of the extension.

Internal extensions are started and stopped by the runtime process, so they are not permitted to register for the Shutdown event.

**Path** – /extension/register

**Method** – POST

**Request headers**

• **Lambda-Extension-Accept-Feature** – Use this to specify optional Extensions features during registration. Required: no. Type: comma separated string. Features available to specify using this setting:
  • **accountId** – If specified, the Extension registration response will contain the account ID associated with the Lambda function that you're registering the Extension for.

**Request body parameters**

• **events** – Array of the events to register for. Required: no. Type: array of strings. Valid strings: INVOKE, SHUTDOWN.

**Response headers**

• **Lambda-Extension-Identifier** – Generated unique agent identifier (UUID string) that is required for all subsequent requests.

**Response codes**

• 200 – Response body contains the function name, function version, and handler name.
• 400 – Bad Request
• 403 – Forbidden
• 500 – Container error. Non-recoverable state. Extension should exit promptly.

**Example Example request body**

```json
{
  'events': ['INVOKE', 'SHUTDOWN']
}
```

**Example Example response body**

```json
{
  "functionName": "helloWorld",
  "functionVersion": "$LATEST",
  "handler": "lambda_function.lambda_handler"
}
```
Example response body with optional accountId feature

```json
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler",
    "accountId": "123456789012"
}
```

Next

Extensions send a Next API request to receive the next event, which can be an Invoke event or a Shutdown event. The response body contains the payload, which is a JSON document that contains event data.

The extension sends a Next API request to signal that it is ready to receive new events. This is a blocking call.

Do not set a timeout on the GET call, as the extension can be suspended for a period of time until there is an event to return.

**Path** – /extension/event/next

**Method** – GET

**Request headers**

- **Lambda-Extension-Identifier** – Unique identifier for extension (UUID string). Required: yes. Type: UUID string.

**Response headers**

- **Lambda-Extension-Event-Identifier** – Unique identifier for the event (UUID string).

**Response codes**

- 200 – Response contains information about the next event (EventInvoke or EventShutdown).
- 403 – Forbidden
**Init error**

The extension uses this method to report an initialization error to Lambda. Call it when the extension fails to initialize after it has registered. After Lambda receives the error, no further API calls succeed. The extension should exit after it receives the response from Lambda.

**Path** – /extension/init/error

**Method** – POST

**Request headers**

- Lambda-Extension-Function-Error-Type – Error type that the extension encountered. Required: yes. This header consists of a string value. Lambda accepts any string, but we recommend a format of `<category.reason>`. For example:
  - Extension.NoSuchHandler
  - Extension.APIKeyNotFound
  - Extension.ConfigInvalid
  - Extension.UnknownReason

**Request body parameters**

- ErrorRequest – Information about the error. Required: no.

This field is a JSON object with the following structure:

```json
{
  errorMessage: string (text description of the error),
  errorType: string,
  stackTrace: array of strings
}
```

Note that Lambda accepts any value for errorType.

The following example shows a Lambda function error message in which the function could not parse the event data provided in the invocation.
Example Function error

```json
{
    "errorMessage" : "Error parsing event data.",
    "errorType" : "InvalidEventDataException",
    "stackTrace": [ ]
}
```

Response codes

- 202 – Accepted
- 400 – Bad Request
- 403 – Forbidden

Exit error

The extension uses this method to report an error to Lambda before exiting. Call it when you encounter an unexpected failure. After Lambda receives the error, no further API calls succeed. The extension should exit after it receives the response from Lambda.

Path – /extension/exit/error

Method – POST

Request headers

- Lambda-Extension-Function-Error-Type – Error type that the extension encountered. Required: yes. This header consists of a string value. Lambda accepts any string, but we recommend a format of <category.reason>. For example:
  - Extension.NoSuchHandler
  - Extension.APIKeyNotFound
  - Extension.ConfigInvalid
  - Extension.UnknownReason
**Request body parameters**

- **ErrorRequest** – Information about the error. Required: no.

This field is a JSON object with the following structure:

```json
{
    "errorMessage": string (text description of the error),
    "errorType": string,
    "stackTrace": array of strings
}
```

Note that Lambda accepts any value for `errorType`.

The following example shows a Lambda function error message in which the function could not parse the event data provided in the invocation.

**Example Function error**

```json
{
    "errorMessage": "Error parsing event data.",
    "errorType": "InvalidEventDataException",
    "stackTrace": []
}
```

**Response codes**

- 202 – Accepted
- 400 – Bad Request
- 403 – Forbidden
Lambda Telemetry API

Using the Lambda Telemetry API, your extensions can directly receive telemetry data from Lambda. During function initialization and invocation, Lambda automatically captures telemetry, such as logs, platform metrics, and platform traces. With Telemetry API, extensions can get this telemetry data directly from Lambda in near real time.

You can subscribe your Lambda extensions to telemetry streams directly from within the Lambda execution environment. After subscribing, Lambda automatically streams all telemetry data to your extensions. You can then process, filter, and deliver that data to your preferred destination, such as an Amazon Simple Storage Service (Amazon S3) bucket or a third-party observability tools provider.

The following diagram shows how the Extensions API and Telemetry API connect extensions to Lambda from within the execution environment. The Runtime API also connects your runtime and function to Lambda.

**Important**

The Lambda Telemetry API supersedes the Lambda Logs API. **While the Logs API remains fully functional, we recommend using only the Telemetry API going forward.** You can subscribe your extension to a telemetry stream using either the Telemetry API or the Logs API. After subscribing using one of these APIs, any attempt to subscribe using the other API returns an error.

Extensions can use the Telemetry API to subscribe to three different telemetry streams:
• **Platform telemetry** – Logs, metrics, and traces, which describe events and errors related to the execution environment runtime lifecycle, extension lifecycle, and function invocations.

• **Function logs** – Custom logs that the Lambda function code generates.

• **Extension logs** – Custom logs that the Lambda extension code generates.

---

**Note**

Lambda sends logs and metrics to CloudWatch, and traces to X-Ray (if you've activated tracing), even if an extension subscribes to telemetry streams.

---

**Sections**

• [Creating extensions using the Telemetry API](#)

• [Registering your extension](#)

• [Creating a telemetry listener](#)

• [Specifying a destination protocol](#)

• [Configuring memory usage and buffering](#)

• [Sending a subscription request to the Telemetry API](#)

• [Inbound Telemetry API messages](#)

• [Lambda Telemetry API reference](#)

• [Lambda Telemetry API Event schema reference](#)

• [Converting Lambda Telemetry API Event objects to OpenTelemetry Spans](#)

• [Lambda Logs API](#)

---

**Creating extensions using the Telemetry API**

Lambda extensions run as independent processes in the execution environment, and can continue to run after the function invocation completes. Because extensions are separate processes, you can write them in a language different from the function code. We recommend implementing extensions using a compiled language such as Golang or Rust. This way, the extension is a self-contained binary that can be compatible with any supported runtime.

The following diagram illustrates a four-step process to create an extension that receives and processes telemetry data using the Telemetry API.
Here is each step in more detail:

1. Register your extension using the the section called “Extensions API”. This provides you with a Lambda-Extension-Identifier, which you'll need in the following steps. For more information about how to register your extension, see the section called “Registering your extension”.

2. Create a telemetry listener. This can be a basic HTTP or TCP server. Lambda uses the URI of the telemetry listener to send telemetry data to your extension. For more information, see the section called “Creating a telemetry listener”.

3. Using the Subscribe API in the Telemetry API, subscribe your extension to your desired telemetry streams. You'll need the URI of your telemetry listener for this step. For more information, see the section called “Sending a subscription request to the Telemetry API”.

4. Get telemetry data from Lambda via the telemetry listener. You can do any custom processing of this data, such as dispatching the data to Amazon S3 or to an external observability service.
Note

A Lambda function's execution environment can start and stop multiple times as part of its lifecycle. In general, your extension code runs during function invocations, and also up to 2 seconds during the shutdown phase. We recommend batching the telemetry as it arrives to your listener, and using the Invoke and Shutdown lifecycle events to dispatch each batch to their desired destinations.

Registering your extension

Before you can subscribe to receive telemetry data, you must register your Lambda extension. Registration occurs during the extension initialization phase. The following example shows an HTTP request to register an extension.

```plaintext
POST http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
Lambda-Extension-Name: lambda_extension_name
{
  'events': [ 'INVOKE', 'SHUTDOWN']
}
```

If the request succeeds, the subscriber receives an HTTP 200 success response. The response header contains the Lambda-Extension-Identifier. The response body contains other properties of the function.

```
HTTP/1.1 200 OK
Lambda-Extension-Identifier: a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
{
  "functionName": "lambda_function",
  "functionVersion": "$LATEST",
  "handler": "lambda_handler",
  "accountId": "123456789012"
}
```

For more information, see the section called “Extensions API reference”.

Creating a telemetry listener

Your Lambda extension must have a listener that handles incoming requests from the Telemetry API. The following code shows an example telemetry listener implementation in Golang:
AWS Lambda

Developer Guide

// Starts the server in a goroutine where the log events will be sent
func (s *TelemetryApiListener) Start() (string, error) {
address := listenOnAddress()
l.Info("[listener:Start] Starting on address", address)
s.httpServer = &http.Server{Addr: address}
http.HandleFunc("/", s.http_handler)
go func() {
err := s.httpServer.ListenAndServe()
if err != http.ErrServerClosed {
l.Error("[listener:goroutine] Unexpected stop on Http Server:", err)
s.Shutdown()
} else {
l.Info("[listener:goroutine] Http Server closed:", err)
}
}()
return fmt.Sprintf("http://%s/", address), nil
}
// http_handler handles the requests coming from the Telemetry API.
// Everytime Telemetry API sends log events, this function will read them from the
response body
// and put into a synchronous queue to be dispatched later.
// Logging or printing besides the error cases below is not recommended if you have
subscribed to
// receive extension logs. Otherwise, logging here will cause Telemetry API to send new
logs for
// the printed lines which may create an infinite loop.
func (s *TelemetryApiListener) http_handler(w http.ResponseWriter, r *http.Request) {
body, err := ioutil.ReadAll(r.Body)
if err != nil {
l.Error("[listener:http_handler] Error reading body:", err)
return
}
// Parse and put the log messages into the queue
var slice []interface{}
_ = json.Unmarshal(body, &slice)
for _, el := range slice {
s.LogEventsQueue.Put(el)
}

Creating a telemetry listener

1594


Specifying a destination protocol

When you subscribe to receive telemetry using the Telemetry API, you can specify a destination protocol in addition to the destination URI:

```
{
    "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
    }
}
```

Lambda accepts two protocols for receiving telemetry:

- **HTTP (recommended)** – Lambda delivers telemetry to a local HTTP endpoint (http://sandbox.localdomain:${PORT}/${PATH}) as an array of records in JSON format. The $PATH parameter is optional. Lambda supports only HTTP, not HTTPS. Lambda delivers telemetry through POST requests.
- **TCP** – Lambda delivers telemetry to a TCP port in Newline delimited JSON (NDJSON) format.

**Note**

We strongly recommend using HTTP rather than TCP. With TCP, the Lambda platform cannot acknowledge when it delivers telemetry to the application layer. Therefore, if your extension crashes, you might lose telemetry. HTTP does not have this limitation.

Before subscribing to receive telemetry, set up the local HTTP listener or TCP port. During setup, note the following:

- Lambda sends telemetry only to destinations that are inside the execution environment.
Lambda retries the attempt to send telemetry (with backoff) if there is no listener, or if the POST request results in an error. If the telemetry listener crashes, then it continues to receive telemetry after Lambda restarts the execution environment.

Lambda reserves port 9001. There are no other port number restrictions or recommendations.

Configuring memory usage and buffering

An execution environment’s memory usage increases linearly as the number of subscribers increases. Subscriptions consume memory resources because each subscription opens a new memory buffer to store telemetry data. Buffer memory usage counts towards overall memory consumption in the execution environment.

When you subscribe to receive telemetry using the Telemetry API, you can buffer telemetry data and deliver it to subscribers in batches. To help optimize memory usage, you can specify a buffering configuration:

```json
{
   "buffering": {
      "maxBytes": 256*1024,
      "maxItems": 1000,
      "timeoutMs": 100
   }
}
```

Buffering configuration settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Defaults and limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxBytes</td>
<td>The maximum volume of telemetry (in bytes) to buffer in memory.</td>
<td>Default: 262,144 Minimum: 262,144 Maximum: 1,048,576</td>
</tr>
<tr>
<td>maxItems</td>
<td>The maximum number of events to buffer in memory.</td>
<td>Default: 10,000 Minimum: 1,000 Maximum: 10,000</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Defaults and limits</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>timeoutMs</td>
<td>The maximum time (in milliseconds) to buffer a batch.</td>
<td>Default: 1,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minimum: 25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Maximum: 30,000</td>
</tr>
</tbody>
</table>

When configuring buffering, note the following points:

- If any of the input streams are closed, then Lambda flushes the logs. This can happen if, for example, the runtime crashes.
- Each subscriber can specify a different buffering configuration in their subscription request.
- Consider the buffer size that you need for reading the data. Expect to receive payloads as large as \( 2 \times \text{maxBytes} + \text{metadataBytes} \), where \( \text{maxBytes} \) is part of your buffering configuration. To get an idea of how much \( \text{metadataBytes} \) you should account for, review the following metadata. Lambda adds metadata similar to this to each record:

  ```json
  {
    "time": "2022-08-20T12:31:32.123Z",
    "type": "function",
    "record": "Hello World"
  }
  ```

- If the subscriber cannot process incoming telemetry fast enough, or if your function code generates very high log volume, Lambda might drop records to keep memory utilization bounded. When this occurs, Lambda sends a `platform.logsDropped` event.

### Sending a subscription request to the Telemetry API

Lambda extensions can subscribe to receive telemetry data by sending a subscription request to the Telemetry API. The subscription request should contain information about the types of events that you want the extension to subscribe to. In addition, the request can contain delivery destination information and a buffering configuration.

Before sending a subscription request, you must have an extension ID (Lambda-Extension-Identifier). When you register your extension with the Extensions API, you obtain an extension ID from the API response.
Subscription occurs during the extension initialization phase. The following example shows an HTTP request to subscribe to all three telemetry streams: platform telemetry, function logs, and extension logs.

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
    "schemaVersion": "2022-12-13",
    "types": [
        "platform",
        "function",
        "extension"
    ],
    "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
    },
    "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
    }
}
```

If the request succeeds, then the subscriber receives an HTTP 200 success response.

```
HTTP/1.1 200 OK
"OK"
```

**Inbound Telemetry API messages**

After subscribing using the Telemetry API, an extension automatically starts to receive telemetry from Lambda via POST requests to the telemetry listener. Each POST request body contains an array of Event objects. Event is a JSON object with the following schema:

```
{
    time: String,
    type: String,
    record: Object
}
```
• The `time` property defines when the Lambda platform generated the event. This isn't the same as when the event actually occurred. The string value of `time` is a timestamp in ISO 8601 format.

• The `type` property defines the event type. The following table describes all possible values.

• The `record` property defines a JSON object that contains the telemetry data. The schema of this JSON object depends on the type.

The following table summarizes all types of Event objects, and links to the Telemetry API Event schema reference for each event type.

**Telemetry API message types**

<table>
<thead>
<tr>
<th>Category</th>
<th>Event type</th>
<th>Description</th>
<th>Event record schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform event</td>
<td><code>platform.initStart</code></td>
<td>Function initialization started.</td>
<td>the section called “platform.initStart” schema</td>
</tr>
<tr>
<td>Platform event</td>
<td><code>platform.initRuntimeDone</code></td>
<td>Function initialization completed.</td>
<td>the section called “platform.initRuntimeDone” schema</td>
</tr>
<tr>
<td>Platform event</td>
<td><code>platform.initReport</code></td>
<td>A report of function initialization.</td>
<td>the section called “platform.initReport” schema</td>
</tr>
<tr>
<td>Platform event</td>
<td><code>platform.start</code></td>
<td>Function invocation started.</td>
<td>the section called “platform.start” schema</td>
</tr>
<tr>
<td>Platform event</td>
<td><code>platform.runtimeDone</code></td>
<td>The runtime finished processing an event with either success or failure.</td>
<td>the section called “platform.runtimeDone” schema</td>
</tr>
</tbody>
</table>

Inbound Telemetry API messages
<table>
<thead>
<tr>
<th>Category</th>
<th>Event type</th>
<th>Description</th>
<th>Event record schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform event</td>
<td>platform.report</td>
<td>A report of function invocation.</td>
<td>the section called “platform.report” schema</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.restoreStart</td>
<td>Runtime restore started.</td>
<td>the section called “platform.restoreStart” schema</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.restoreRuntimeDone</td>
<td>Runtime restore completed.</td>
<td>the section called “platform.restoreRuntimeDone” schema</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.restoreReport</td>
<td>Report of runtime restore.</td>
<td>the section called “platform.restoreReport” schema</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.telemetrySubscription</td>
<td>The extension subscribed to the Telemetry API.</td>
<td>the section called “platform.telemetrySubscription” schema</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.logsDropped</td>
<td>Lambda dropped log entries.</td>
<td>the section called “platform.logsDropped” schema</td>
</tr>
<tr>
<td>Function logs</td>
<td>function</td>
<td>A log line from function code.</td>
<td>the section called “function” schema</td>
</tr>
<tr>
<td>Category</td>
<td>Event type</td>
<td>Description</td>
<td>Event record schema</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------</td>
<td>----------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>Extension logs</td>
<td>extension</td>
<td>A log line from extension code.</td>
<td>the section called &quot;extension&quot; schema</td>
</tr>
</tbody>
</table>
Lambda Telemetry API reference

Use the Lambda Telemetry API endpoint to subscribe extensions to telemetry streams. You can retrieve the Telemetry API endpoint from the AWS_LAMBDA_RUNTIME_API environment variable. To send an API request, append the API version (2022-07-01/) and telemetry/. For example:

http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/

For the OpenAPI Specification (OAS) definition of the subscription responses version 2022-12-13, see the following:

- HTTP – telemetry-api-http-schema.zip
- TCP – telemetry-api-tcp-schema.zip

API operations

- Subscribe

Subscribe

To subscribe to a telemetry stream, a Lambda extension can send a Subscribe API request.

- Path – /telemetry
- Method – PUT
- Headers
  - Content-Type: application/json
- Request body parameters
  - schemaVersion
    - Required: Yes
    - Type: String
    - Valid values: "2022-12-13" or "2022-07-01"
  - destination – The configuration settings that define the telemetry event destination and the protocol for event delivery.
    - Required: Yes
    - Type: Object
• **protocol** – The protocol that Lambda uses to send telemetry data.
  • Required: Yes
  • Type: String
  • Valid values: "HTTP" | "TCP"

• **URI** – The URI to send telemetry data to.
  • Required: Yes
  • Type: String

  For more information, see [the section called “Specifying a destination protocol”](#).

• **types** – The types of telemetry that you want the extension to subscribe to.
  • Required: Yes
  • Type: Array of strings
  • Valid values: "platform" | "function" | "extension"

• **buffering** – The configuration settings for event buffering.
  • Required: No
  • Type: Object

```json
{
    "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
    }
}
```

• **maxItems** – The maximum number of events to buffer in memory.
  • Required: No
  • Type: Integer
  • Default: 1,000
  • Minimum: 1,000
• Maximum: 10,000

• **maxBytes** – The maximum volume of telemetry (in bytes) to buffer in memory.
  • Required: No
  • Type: Integer
  • Default: 262,144
  • Minimum: 262,144
  • Maximum: 1,048,576

• **timeoutMs** – The maximum time (in milliseconds) to buffer a batch.
  • Required: No
  • Type: Integer
  • Default: 1,000
  • Minimum: 25
  • Maximum: 30,000

• For more information, see the section called “Configuring memory usage and buffering”.

**Example Subscribe API request**

```json
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2022-12-13",
   "types": [
       "platform",
       "function",
       "extension"
   ],
   "buffering": {
       "maxItems": 1000,
       "maxBytes": 256*1024,
       "timeoutMs": 100
   },
   "destination": {
       "protocol": "HTTP",
       "URI": "http://sandbox.localdomain:8080"
   }
}
```
If the Subscribe request succeeds, the extension receives an HTTP 200 success response:

```
HTTP/1.1 200 OK
"OK"
```

If the Subscribe request fails, the extension receives an error response. For example:

```
HTTP/1.1 400 OK
{
    "errorType": "ValidationError",
    "errorMessage": "URI port is not provided; types should not be empty"
}
```

Here are some additional response codes that the extension can receive:

- 200 – Request completed successfully
- 202 – Request accepted. Subscription request response in local testing environment
- 400 – Bad request
- 500 – Service error
Lambda Telemetry API Event schema reference

Use the Lambda Telemetry API endpoint to subscribe extensions to telemetry streams. You can retrieve the Telemetry API endpoint from the `AWS_LAMBDA_RUNTIME_API` environment variable. To send an API request, append the API version (`2022-07-01/`) and `telemetry/`. For example:

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

For the OpenAPI Specification (OAS) definition of the subscription responses version `2022-12-13`, see the following:

- HTTP – [telemetry-api-http-schema.zip](#)
- TCP – [telemetry-api-tcp-schema.zip](#)

The following table is a summary of all the types of Event objects that the Telemetry API supports.

### Telemetry API message types

<table>
<thead>
<tr>
<th>Category</th>
<th>Event type</th>
<th>Description</th>
<th>Event record schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform event</td>
<td>platform.initStart</td>
<td>Function initialization started.</td>
<td>the section called “platform.initStart” schema</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.initRuntimeDone</td>
<td>Function initialization completed.</td>
<td>the section called “platform.initRuntimeDone” schema</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.initReport</td>
<td>A report of function initialization.</td>
<td>the section called “platform.initReport” schema</td>
</tr>
<tr>
<td>Category</td>
<td>Event type</td>
<td>Description</td>
<td>Event record schema</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.start</td>
<td>Function invocation started.</td>
<td>the section called &quot;platform.start&quot; schema</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.runtimeDone</td>
<td>The runtime finished processing an event with either success or failure.</td>
<td>the section called &quot;platform.runtimeDone&quot; schema</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.report</td>
<td>A report of function invocation.</td>
<td>the section called &quot;platform.report&quot; schema</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.restoreStart</td>
<td>Runtime restore started.</td>
<td>the section called &quot;platform.restoreStart&quot; schema</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.restoreRuntimeDone</td>
<td>Runtime restore completed.</td>
<td>the section called &quot;platform.restoreRuntimeDone&quot; schema</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.restoreReport</td>
<td>Report of runtime restore.</td>
<td>the section called &quot;platform.restoreReport&quot; schema</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.telemetry</td>
<td>The extension subscribed to the Telemetry API.</td>
<td>the section called &quot;platform.telemetry&quot; schema</td>
</tr>
<tr>
<td></td>
<td>Subscription</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Event type</td>
<td>Description</td>
<td>Event record schema</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------</td>
<td>------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Platform event</td>
<td>platform.logsDropped</td>
<td>Lambda dropped log entries.</td>
<td>the section called “platform.logsDropped” schema</td>
</tr>
<tr>
<td>Function logs</td>
<td>function</td>
<td>A log line from function code.</td>
<td>the section called “function” schema</td>
</tr>
<tr>
<td>Extension logs</td>
<td>extension</td>
<td>A log line from extension code.</td>
<td>the section called “extension” schema</td>
</tr>
</tbody>
</table>

Contents

- Telemetry API Event object types
  - platform.initStart
  - platform.initRuntimeDone
  - platform.initReport
  - platform.start
  - platform.runtimeDone
  - platform.report
  - platform.restoreStart
  - platform.restoreRuntimeDone
  - platform.restoreReport
  - platform.extension
  - platform.telemetrySubscription
  - platform.logsDropped
- function
- extension
- Shared object types
  - InitPhase
- InitReportMetrics
Telemetry API Event object types

This section details the types of Event objects that the Lambda Telemetry API supports. In the event descriptions, a question mark (?) indicates that the attribute may not be present in the object.

**platform.initStart**

A platform.initStart event indicates that the function initialization phase has started. A platform.initStart Event object has the following shape:

```
Event: Object
- time: String
- type: String = platform.initStart
- record: PlatformInitStart
```

The PlatformInitStart object has the following attributes:

- **functionName** – String
- **functionVersion** – String
- **initializationType** – the section called “InitType” object
- **instanceId?** – String
- **instanceMaxMemory?** – Integer
- **phase** – the section called “InitPhase” object
- **runtimeVersion?** – String
- **runtimeVersionArn?** – String
The following is an example Event of type platform.initStart:

```json
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.initStart",
    "record": {
        "initializationType": "on-demand",
        "phase": "init",
        "runtimeVersion": "nodejs-14.v3",
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

**platform.initRuntimeDone**

A platform.initRuntimeDone event indicates that the function initialization phase has completed. A platform.initRuntimeDone Event object has the following shape:

- Event: Object
  - time: String
  - type: String = platform.initRuntimeDone
  - record: PlatformInitRuntimeDone

The PlatformInitRuntimeDone object has the following attributes:

- **initializationType** – the section called “InitType” object
- **phase** – the section called “InitPhase” object
- **status** – the section called “Status” object
- **spans?** – List of the section called “Span” objects

The following is an example Event of type platform.initRuntimeDone:

```json
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initRuntimeDone",
    "record": {
```
platform.initReport

A platform.initReport event contains an overall report of the function initialization phase. A platform.initReport Event object has the following shape:

```json
Event: Object
- time: String
- type: String = platform.initReport
- record: PlatformInitReport
```

The PlatformInitReport object has the following attributes:

- **errorType?** – string
- **initializationType** – the section called “InitType” object
- **phase** – the section called “InitPhase” object
- **metrics** – the section called “InitReportMetrics” object
- **spans?** – List of the section called “Span” objects
- **status** – the section called “Status” object

The following is an example Event of type platform.initReport:

```json
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initReport",
    "record": {
        "initializationType": "on-demand",
        "status": "success",
    }
}
```
"phase": "init",
"metrics": {
   "durationMs": 125.33
},
"spans": [
   {
      "name": "someTimeSpan",
      "start": "2022-06-02T12:02:33.913Z",
      "durationMs": 90.1
   }
]
}

**platform.start**

A platform.start event indicates that the function invocation phase has started. A platform.start Event object has the following shape:

```json
Event: Object
- time: String
- type: String = platform.start
- record: PlatformStart
```

The PlatformStart object has the following attributes:

- **requestId** – String
- **version?** – String
- **tracing?** – [the section called “TraceContext”](#)

The following is an example Event of type platform.start:

```json
{
   "time": "2022-10-12T00:00:15.064Z",
   "type": "platform.start",
   "record": {
      "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
      "version": "$LATEST",
      "tracing": {
         "spanId": "54565fb41ac79632",
         "type": "X-Amzn-Trace-Id",
```

---

**Event schema reference**

1612
platform.runtimeDone

A platform.runtimeDone event indicates that the function invocation phase has completed. A platform.runtimeDone Event object has the following shape:

Event: Object
- time: String
- type: String = platform.runtimeDone
- record: PlatformRuntimeDone

The PlatformRuntimeDone object has the following attributes:

- **errorType?** – String
- **metrics?** – the section called “RuntimeDoneMetrics” object
- **requestId** – String
- **status** – the section called “Status” object
- **spans?** – List of the section called “Span” objects
- **tracing?** – the section called “TraceContext” object

The following is an example Event of type platform.runtimeDone:

```json
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.runtimeDone",
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
        "status": "success",
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
            "value": "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        }
    }
}
```
"spans": [
  {
    "name": "someTimeSpan",
    "start": "2022-08-02T12:01:23:521Z",
    "durationMs": 80.0
  },
  {
    "metrics": {
      "durationMs": 140.0,
      "producedBytes": 16
    }
  }
],
"metrics": {
  "durationMs": 140.0,
  "producedBytes": 16
}
}

platform.report

A platform.report event contains an overall report of the function initialization phase. A platform.report Event object has the following shape:

Event: Object
- time: String
- type: String = platform.report
- record: PlatformReport

The PlatformReport object has the following attributes:

- **metrics** – the section called “ReportMetrics” object
- **requestId** – String
- **spans?** – List of the section called “Span” objects
- **status** – the section called “Status” object
- **tracing?** – the section called “TraceContext” object

The following is an example Event of type platform.report:

```json
{
  "time": "2022-10-12T00:01:15.000Z",
  "type": "platform.report",
  "record": {
    "metrics": {
```
platform.restoreStart

A `platform.restoreStart` event indicates that a function environment restoration event started. In an environment restoration event, Lambda creates the environment from a cached snapshot rather than initializing it from scratch. For more information, see Lambda SnapStart. A `platform.restoreStart` Event object has the following shape:

```
Event: Object
- time: String
- type: String = platform.restoreStart
- record: PlatformRestoreStart
```

The `PlatformRestoreStart` object has the following attributes:

- **functionName** – String
- **functionVersion** – String
- **instanceId?** – String
- **instanceMaxMemory?** – String
- **runtimeVersion?** – String
- **runtimeVersionArn?** – String

The following is an example Event of type `platform.restoreStart`:

```
{
  "time": "2022-10-12T00:00:15.064Z",
  "type": "platform.restoreStart",
  "record": {
    "runtimeVersion": "nodejs-14.v3",
    "runtimeVersionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:2022-10-12T00:00:15.064Z",
    "instanceId": "i-01234567890123456",
    "instanceMaxMemory": "128 MB",
    "functionName": "my-function",
    "functionVersion": "2022-10-12T00:00:15.064Z"
  }
}
```
platform.restoreRuntimeDone

A platform.restoreRuntimeDone event indicates that a function environment restoration event completed. In an environment restoration event, Lambda creates the environment from a cached snapshot rather than initializing it from scratch. For more information, see Lambda SnapStart. A platform.restoreRuntimeDone Event object has the following shape:

Event: Object
- time: String
- type: String = platform.restoreRuntimeDone
- record: PlatformRestoreRuntimeDone

The PlatformRestoreRuntimeDone object has the following attributes:

- **errorType?** – String
- **spans?** – List of the section called “Span” objects
- **status** – the section called “Status” object

The following is an example Event of type platform.restoreRuntimeDone:

```json
{
  "time": "2022-10-12T00:00:15.064Z",
  "type": "platform.restoreRuntimeDone",
  "record": {
    "status": "success",
    "spans": [
      {
        "name": "someTimeSpan",
        "start": "2022-08-02T12:01:23:521Z",
        "durationMs": 80.0
      }
    ]
  }
}
```
platform.restoreReport

A platform.restoreReport event contains an overall report of a function restoration event. A platform.restoreReport Event object has the following shape:

```
Event: Object
  - time: String
  - type: String = platform.restoreReport
  - record: PlatformRestoreReport
```

The PlatformRestoreReport object has the following attributes:

- **errorType?** – string
- **metrics?** – the section called “RestoreReportMetrics” object
- **spans?** – List of the section called “Span” objects
- **status** – the section called “Status” object

The following is an example Event of type platform.restoreReport:

```
{
  "time": "2022-10-12T00:00:15.064Z",
  "type": "platform.restoreReport",
  "record": {
    "status": "success",
    "metrics": {
      "durationMs": 15.19
    },
    "spans": [
      {
        "name": "someTimeSpan",
        "start": "2022-08-02T12:01:23:521Z",
        "durationMs": 30.0
      }
    ]
  }
}
```
**platform.extension**

An extension event contains logs from the extension code. An extension Event object has the following shape:

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

The PlatformExtension object has the following attributes:

- **events** – List of String
- **name** – String
- **state** – String

The following is an example Event of type platform.extension:

```
{
   "time": "2022-10-12T00:02:15.000Z",
   "type": "platform.extension",
   "record": {
      "events": [ "INVOKE", "SHUTDOWN" ],
      "name": "my-telemetry-extension",
      "state": "Ready"
   }
}
```

**platform.telemetrySubscription**

A platform.telemetrySubscription event contains information about an extension subscription. A platform.telemetrySubscription Event object has the following shape:

```
Event: Object
- time: String
- type: String = platform.telemetrySubscription
- record: PlatformTelemetrySubscription
```

The PlatformTelemetrySubscription object has the following attributes:
• **name** – String
• **state** – String
• **types** – List of String

The following is an example Event of type `platform.telemetrySubscription`:

```json
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.telemetrySubscription",
    "record": {
        "name": "my-telemetry-extension",
        "state": "Subscribed",
        "types": [ "platform", "function" ]
    }
}
```

**platform.logsDropped**

A `platform.logsDropped` event contains information about dropped events. Lambda emits the `platform.logsDropped` event when an extension can't process one or more events. A `platform.logsDropped` Event object has the following shape:

<table>
<thead>
<tr>
<th>Event: Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>- time: String</td>
</tr>
<tr>
<td>- type: String = <code>platform.logsDropped</code></td>
</tr>
<tr>
<td>- record: PlatformLogsDropped</td>
</tr>
</tbody>
</table>

The `PlatformLogsDropped` object has the following attributes:

• **droppedBytes** – Integer
• **droppedRecords** – Integer
• **reason** – String

The following is an example Event of type `platform.logsDropped`:

```json
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.logsDropped",
    "record": {
```
"droppedBytes": 12345,
"droppedRecords": 123,
"reason": "Consumer seems to have fallen behind as it has not acknowledged receipt of logs."
}
}

**function**

A function event contains logs from the function code. A function Event object has the following shape:

```javascript
Event: Object
- time: String
- type: String = function
- record: {}  
```

The format of the record field depends on whether your function's logs are formatted in plain text or JSON format. To learn more about log format configuration options, see the section called “Configuring JSON and plain text log formats”

The following is an example Event of type function where the log format is plain text:

```json
{
  "time": "2022-10-12T00:03:50.000Z",
  "type": "function",
  "record": "[INFO] Hello world, I am a function!"
}
```

The following is an example Event of type function where the log format is JSON:

```json
{
  "time": "2022-10-12T00:03:50.000Z",
  "type": "function",
  "record": {
    "timestamp": "2022-10-12T00:03:50.000Z",
    "level": "INFO",
    "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
    "message": "Hello world, I am a function!"
  }
}
```
Note

If the schema version you’re using is older than the 2022-12-13 version, then the "record" is always rendered as a string even when your function’s logging format is configured as JSON.

event

A event contains logs from the event code. A event object has the following shape:

```
Event: Object
    - time: String
    - type: String = extension
    - record: {}
```

The format of the record field depends on whether your function’s logs are formatted in plain text or JSON format. To learn more about log format configuration options, see the section called “Configuring JSON and plain text log formats”.

The following is an example Event of type extension where the log format is plain text:

```
{
    "time": "2022-10-12T03:50.000Z",
    "type": "extension",
    "record": "[INFO] Hello world, I am an extension!"
}
```

The following is an example Event of type extension where the log format is JSON:

```
{
    "time": "2022-10-12T03:50.000Z",
    "type": "extension",
    "record": {
        "timestamp": "2022-10-12T03:50.000Z",
        "level": "INFO",
        "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
        "message": "Hello world, I am an extension!"
    }
}
```
Shared object types

This section details the types of shared objects that the Lambda Telemetry API supports.

InitPhase

A string enum that describes the phase when the initialization step occurs. In most cases, Lambda runs the function initialization code during the init phase. However, in some error cases, Lambda may re-run the function initialization code during the invoke phase. (This is called a suppressed init.)

- **Type** – String
- **Valid values** – init|invoke|snap-start

InitReportMetrics

An object that contains metrics about an initialization phase.

- **Type** – Object

An InitReportMetrics object has the following shape:

```
InitReportMetrics: Object
  - durationMs: Double
```

The following is an example InitReportMetrics object:

```
{
  "durationMs": 247.88
}
```
**InitType**

A string enum that describes how Lambda initialized the environment.

- **Type** – String
- **Valid values** – on-demand|provisioned-concurrency

**ReportMetrics**

An object that contains metrics about a completed phase.

- **Type** – Object

A ReportMetrics object has the following shape:

```json
ReportMetrics: Object
- billedDurationMs: Integer
- durationMs: Double
- initDurationMs?: Double
- maxMemoryUsedMB: Integer
- memorySizeMB: Integer
- restoreDurationMs?: Double
```

The following is an example ReportMetrics object:

```json
{
    "billedDurationMs": 694,
    "durationMs": 693.92,
    "initDurationMs": 397.68,
    "maxMemoryUsedMB": 84,
    "memorySizeMB": 128
}
```

**RestoreReportMetrics**

An object that contains metrics about a completed restoration phase.

- **Type** – Object
A RestoreReportMetrics object has the following shape:

```plaintext
RestoreReportMetrics: Object
- durationMs: Double
```

The following is an example RestoreReportMetrics object:

```json
{
    "durationMs": 15.19
}
```

**RuntimeDoneMetrics**

An object that contains metrics about an invocation phase.

- **Type** – Object

A RuntimeDoneMetrics object has the following shape:

```plaintext
RuntimeDoneMetrics: Object
- durationMs: Double
- producedBytes?: Integer
```

The following is an example RuntimeDoneMetrics object:

```json
{
    "durationMs": 200.0,
    "producedBytes": 15
}
```

**Span**

An object that contains details about a span. A span represents a unit of work or operation in a trace. For more information about spans, see [Span](#) on the [Tracing API](#) page of the OpenTelemetry Docs website.

Lambda supports the following spans for the `platform.RuntimeDone` event:

- The `responseLatency` span describes how long it took your Lambda function to start sending the response.
The responseDuration span describes how long it took your Lambda function to finish sending the entire response.

The runtimeOverhead span describes how long it took the Lambda runtime to signal that it is ready to process the next function invoke. This is how long the runtime took to call the next invocation API to get the next event after returning your function response.

The following is an example responseLatency span object:

```
{
    "name": "responseLatency",
    "start": "2022-08-02T12:01:23.521Z",
    "durationMs": 23.02
}
```

**Status**

An object that describes the status of an initialization or invocation phase. If the status is either failure or error, then the Status object also contains an errorType field describing the error.

- **Type** – Object
- **Valid status values** – success|failure|error|timeout

**TraceContext**

An object that describes the properties of a trace.

- **Type** – Object

A TraceContext object has the following shape:

```
TraceContext: Object
- spanId?: String
- type: TracingType enum
- value: String
```

The following is an example TraceContext object:

```
{
```

Event schema reference
TracingType

A string enum that describes the type of tracing in a the section called “TraceContext” object.

- **Type** – String
- **Valid values** – X-Amzn-Trace-Id
Converting Lambda Telemetry API Event objects to OpenTelemetry Spans

The AWS Lambda Telemetry API schema is semantically compatible with OpenTelemetry (OTel). This means that you can convert your AWS Lambda Telemetry API Event objects to OpenTelemetry (OTel) Spans. When converting, you shouldn't map a single Event object to a single OTel Span. Instead, you should present all three events related to a lifecycle phase in a single OTel Span. For example, the start, runtimeDone, and runtimeReport events represent a single function invocation. Present all three of these events as a single OTel Span.

You can convert your events using Span Events or Child (nested) Spans. The tables on this page describe the mappings between Telemetry API schema properties and OTel Span properties for both approaches. For more information about OTel Spans, see Span on the Tracing API page of the OpenTelemetry Docs website.

Sections

- Map to OTel Spans with Span Events
- Map to OTel Spans with Child Spans

Map to OTel Spans with Span Events

In the following tables, e represents the event coming from the telemetry source.

Mapping the *Start events

<table>
<thead>
<tr>
<th>OpenTelemetry</th>
<th>Lambda Telemetry API schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>Span.Name</td>
<td>Your extension generates this value based on the type field.</td>
</tr>
<tr>
<td>Span.StartTime</td>
<td>Use e.time.</td>
</tr>
<tr>
<td>Span.EndTime</td>
<td>N/A, because the event hasn't completed yet.</td>
</tr>
<tr>
<td>Span.Kind</td>
<td>Set to Server.</td>
</tr>
<tr>
<td>Span.Status</td>
<td>Set to Unset.</td>
</tr>
</tbody>
</table>
### OpenTelemetry | Lambda Telemetry API schema

| **Span.TraceId** | Parse the AWS X-Ray header found in `e.tracing.value`, then use the TraceId value. |
| **Span.ParentId** | Parse the X-Ray header found in `e.tracing.value`, then use the Parent value. |
| **Span.SpanId** | Use `e.tracing.spanId` if available. Otherwise, generate a new SpanId. |
| **Span.SpanContext.TraceState** | N/A for an X-Ray trace context. |
| **Span.SpanContext.TraceFlags** | Parse the X-Ray header found in `e.tracing.value`, then use the Sampled value. |
| **Span.Attributes** | Your extension can add any custom values here. |

### Mapping the *RuntimeDone events*

<table>
<thead>
<tr>
<th><strong>OpenTelemetry</strong></th>
<th>Lambda Telemetry API schema</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Span.Name</strong></td>
<td>Your extension generates the value based on the type field.</td>
</tr>
<tr>
<td><strong>Span.StartTime</strong></td>
<td>Use <code>e.time</code> from the matching *Start event. Alternatively, use <code>e.time - e.metrics.durationMs</code></td>
</tr>
<tr>
<td><strong>Span.EndTime</strong></td>
<td>N/A, because the event hasn't completed yet.</td>
</tr>
<tr>
<td><strong>Span.Kind</strong></td>
<td>Set to Server.</td>
</tr>
<tr>
<td><strong>Span.Status</strong></td>
<td>If <code>e.status</code> doesn't equal success, then set to Error.</td>
</tr>
<tr>
<td>OpenTelemetry</td>
<td>Lambda Telemetry API schema</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Span.Events[]</td>
<td>Use <code>e.spans[]</code></td>
</tr>
<tr>
<td>Span.Events[i].Name</td>
<td>Use <code>e.spans[i].name</code></td>
</tr>
<tr>
<td>Span.Events[i].Time</td>
<td>Use <code>e.spans[i].start</code></td>
</tr>
<tr>
<td>Span.TraceId</td>
<td>Parse the AWS X-Ray header found in <code>e.tracing.value</code>, then use the TraceId value.</td>
</tr>
<tr>
<td>Span.ParentId</td>
<td>Parse the X-Ray header found in <code>e.tracing.value</code>, then use the Parent value.</td>
</tr>
<tr>
<td>Span.SpanId</td>
<td>Use the same SpanId from the *Start event. If unavailable, then use <code>e.tracing.spanId</code>, or generate a new SpanId.</td>
</tr>
<tr>
<td>Span.SpanContext.TraceState</td>
<td>N/A for an X-Ray trace context.</td>
</tr>
<tr>
<td>Span.SpanContext.TraceFlags</td>
<td>Parse the X-Ray header found in <code>e.tracing.value</code>, then use the Sampled value.</td>
</tr>
<tr>
<td>Span.Attributes</td>
<td>Your extension can add any custom values here.</td>
</tr>
</tbody>
</table>

### Mapping the *Report events

<table>
<thead>
<tr>
<th>OpenTelemetry</th>
<th>Lambda Telemetry API schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>Span.Name</td>
<td>Your extension generates the value based on the type field.</td>
</tr>
<tr>
<td>Span.StartTime</td>
<td>Use <code>e.time</code> from the matching *Start event. Alternatively, use <code>e.time - e.metrics.durationMs</code></td>
</tr>
</tbody>
</table>

Converting events to OTel Spans
Map to OTel Spans with Child Spans

The following table describes how to convert Lambda Telemetry API events into OTel Spans with Child (nested) Spans for *RuntimeDone Spans. For *Start and *Report mappings, refer to the tables in the section called "Map to OTel Spans with Span Events", as they're the same for Child Spans. In this table, e represents the event coming from the telemetry source.

<table>
<thead>
<tr>
<th>OpenTelemetry</th>
<th>Lambda Telemetry API schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>Span.EndTime</td>
<td>Use e.time.</td>
</tr>
<tr>
<td>Span.Kind</td>
<td>Set to Server.</td>
</tr>
<tr>
<td>Span.Status</td>
<td>Use the same value as the *RuntimeDone event.</td>
</tr>
<tr>
<td>Span.TraceId</td>
<td>Parse the AWS X-Ray header found in e.tracing.value, then use the TraceId value.</td>
</tr>
<tr>
<td>Span.ParentId</td>
<td>Parse the X-Ray header found in e.tracing.value, then use the Parent value.</td>
</tr>
<tr>
<td>Span.SpanId</td>
<td>Use the same SpanId from the *Start event. If unavailable, then use e.tracing.spanId, or generate a new SpanId.</td>
</tr>
<tr>
<td>Span.SpanContext.TraceState</td>
<td>N/A for an X-Ray trace context.</td>
</tr>
<tr>
<td>Span.SpanContext.TraceFlags</td>
<td>Parse the X-Ray header found in e.tracing.value, then use the Sampled value.</td>
</tr>
<tr>
<td>Span.Attributes</td>
<td>Your extension can add any custom values here.</td>
</tr>
</tbody>
</table>
### Mapping the *RuntimeDone events

<table>
<thead>
<tr>
<th>OpenTelemetry</th>
<th>Lambda Telemetry API schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>Span.Name</td>
<td>Your extension generates the value based on the type field.</td>
</tr>
<tr>
<td>Span.StartTime</td>
<td>Use <code>e.time</code> from the matching *Start event.</td>
</tr>
<tr>
<td></td>
<td>Alternatively, use <code>e.time - e.metrics.durationMs</code></td>
</tr>
<tr>
<td>Span.EndTime</td>
<td>N/A, because the event hasn't completed yet.</td>
</tr>
<tr>
<td>Span.Kind</td>
<td>Set to Server.</td>
</tr>
<tr>
<td>Span.Status</td>
<td>If <code>e.status</code> doesn't equal success, then set to Error.</td>
</tr>
<tr>
<td></td>
<td>Otherwise, set to Ok.</td>
</tr>
<tr>
<td>Span.TraceId</td>
<td>Parse the AWS X-Ray header found in <code>e.tracing.value</code>, then use the <code>TraceId</code> value.</td>
</tr>
<tr>
<td>Span.ParentId</td>
<td>Parse the X-Ray header found in <code>e.tracing.value</code>, then use the <code>Parent</code> value.</td>
</tr>
<tr>
<td>Span.SpanId</td>
<td>Use the same <code>SpanId</code> from the *Start event. If unavailable, then use <code>e.tracing.spanId</code>,</td>
</tr>
<tr>
<td></td>
<td>or generate a new <code>SpanId</code>.</td>
</tr>
<tr>
<td>Span.SpanContext.TraceState</td>
<td>N/A for an X-Ray trace context.</td>
</tr>
<tr>
<td>Span.SpanContext.TraceFlags</td>
<td>Parse the X-Ray header found in <code>e.tracing.value</code>, then use the <code>Sampled</code> value.</td>
</tr>
<tr>
<td>Span.Attributes</td>
<td>Your extension can add any custom values here.</td>
</tr>
<tr>
<td>ChildSpan[i].Name</td>
<td>Use <code>e.spans[i].name</code></td>
</tr>
<tr>
<td>OpenTelemetry</td>
<td>Lambda Telemetry API schema</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td>ChildSpan[i].StartTime</td>
<td>Use e.spans[i].start .</td>
</tr>
<tr>
<td>ChildSpan[i].EndTime</td>
<td>Use e.spans[i].start + e.spans[i].durations .</td>
</tr>
<tr>
<td>ChildSpan[i].Kind</td>
<td>Same as parent Span.Kind .</td>
</tr>
<tr>
<td>ChildSpan[i].Status</td>
<td>Same as parent Span.Status .</td>
</tr>
<tr>
<td>ChildSpan[i].TraceId</td>
<td>Same as parent Span.TraceId .</td>
</tr>
<tr>
<td>ChildSpan[i].ParentId</td>
<td>Use parent Span.SpanId .</td>
</tr>
<tr>
<td>ChildSpan[i].SpanId</td>
<td>Generate a new SpanId.</td>
</tr>
<tr>
<td>ChildSpan[i].SpanContext.TraceState</td>
<td>N/A for an X-Ray trace context.</td>
</tr>
<tr>
<td>ChildSpan[i].SpanContext.TraceFlags</td>
<td>Same as parent Span.SpanContext.TraceFlags .</td>
</tr>
</tbody>
</table>
Lambda Logs API

⚠️ Important

The Lambda Telemetry API supersedes the Lambda Logs API. **While the Logs API remains fully functional, we recommend using only the Telemetry API going forward.** You can subscribe your extension to a telemetry stream using either the Telemetry API or the Logs API. After subscribing using one of these APIs, any attempt to subscribe using the other API returns an error.

Lambda automatically captures runtime logs and streams them to Amazon CloudWatch. This log stream contains the logs that your function code and extensions generate, and also the logs that Lambda generates as part of the function invocation.

**Lambda extensions** can use the Lambda Runtime Logs API to subscribe to log streams directly from within the Lambda **execution environment**. Lambda streams the logs to the extension, and the extension can then process, filter, and send the logs to any preferred destination.

The Logs API allows extensions to subscribe to three different logs streams:

- Function logs that the Lambda function generates and writes to *stdout* or *stderr*.
- Extension logs that extension code generates.
- Lambda platform logs, which record events and errors related to invocations and extensions.
Note
Lambda sends all logs to CloudWatch, even when an extension subscribes to one or more of the log streams.

Topics
- Subscribing to receive logs
- Memory usage
- Destination protocols
- Buffering configuration
- Example subscription
- Sample code for Logs API
- Logs API reference
- Log messages

Subscribing to receive logs

A Lambda extension can subscribe to receive logs by sending a subscription request to the Logs API.

To subscribe to receive logs, you need the extension identifier (Lambda-Extension-Identifier). First register the extension to receive the extension identifier. Then subscribe to the Logs API during initialization. After the initialization phase completes, Lambda does not process subscription requests.

Note
Logs API subscription is idempotent. Duplicate subscribe requests do not result in duplicate subscriptions.

Memory usage

Memory usage increases linearly as the number of subscribers increases. Subscriptions consume memory resources because each subscription opens a new memory buffer to store the logs. To help
optimize memory usage, you can adjust the buffering configuration. Buffer memory usage counts towards overall memory consumption in the execution environment.

**Destination protocols**

You can choose one of the following protocols to receive the logs:

1. **HTTP** (recommended) – Lambda delivers logs to a local HTTP endpoint (http://sandbox.localdomain:${PORT}/${PATH}) as an array of records in JSON format. The $PATH parameter is optional. Note that only HTTP is supported, not HTTPS. You can choose to receive logs through PUT or POST.

2. **TCP** – Lambda delivers logs to a TCP port in Newline delimited JSON (NDJSON) format.

We recommend using HTTP rather than TCP. With TCP, the Lambda platform cannot acknowledge when it delivers logs to the application layer. Therefore, you might lose logs if your extension crashes. HTTP does not share this limitation.

We also recommend setting up the local HTTP listener or the TCP port before subscribing to receive logs. During setup, note the following:

- Lambda sends logs only to destinations that are inside the execution environment.
- Lambda retries the attempt to send the logs (with backoff) if there is no listener, or if the POST or PUT request results in an error. If the log subscriber crashes, it continues to receive logs after Lambda restarts the execution environment.
- Lambda reserves port 9001. There are no other port number restrictions or recommendations.

**Buffering configuration**

Lambda can buffer logs and deliver them to the subscriber. You can configure this behavior in the subscription request by specifying the following optional fields. Note that Lambda uses the default value for any field that you do not specify.

- **timeoutMs** – The maximum time (in milliseconds) to buffer a batch. Default: 1,000. Minimum: 25. Maximum: 30,000.
• **maxItems** – The maximum number of events to buffer in memory. Default: 10,000. Minimum: 1,000. Maximum: 10,000.

During buffering configuration, note the following points:

• Lambda flushes the logs if any of the input streams are closed, for example, if the runtime crashes.

• Each subscriber can specify a different buffering configuration in their subscription request.

• Consider the buffer size that you need for reading the data. Expect to receive payloads as large as $2 \times \text{maxBytes} + \text{metadata}$, where maxBytes is configured in the subscribe request. For example, Lambda adds the following metadata bytes to each record:

```json
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "function",
    "record": "Hello World"
}
```

• If the subscriber cannot process incoming logs quickly enough, Lambda might drop logs to keep memory utilization bounded. To indicate the number of dropped records, Lambda sends a `platform.logsDropped` log.

**Example subscription**

The following example shows a request to subscribe to the platform and function logs.

```bash
PUT http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs HTTP/1.1
{  "schemaVersion": "2020-08-15",
   "types": [
      "platform",
      "function"
   ],
   "buffering": {
      "maxItems": 1000,
      "maxBytes": 262144,
      "timeoutMs": 100
   },
   "destination": {
      "protocol": "HTTP",
      "URI": "http://sandbox.localdomain:8080/lambda_logs"
   }
}
```
If the request succeeds, the subscriber receives an HTTP 200 success response.

HTTP/1.1 200 OK
"OK"

Sample code for Logs API

For sample code showing how to send logs to a custom destination, see Using AWS Lambda extensions to send logs to custom destinations on the AWS Compute Blog.

For Python and Go code examples showing how to develop a basic Lambda extension and subscribe to the Logs API, see AWS Lambda Extensions on the AWS Samples GitHub repository. For more information about building a Lambda extension, see the section called “Extensions API”.

Logs API reference

You can retrieve the Logs API endpoint from the AWS_LAMBDA_RUNTIME_API environment variable. To send an API request, use the prefix 2020-08-15/ before the API path. For example:

http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs

The OpenAPI specification for the Logs API version 2020-08-15 is available here: logs-api-request.zip

Subscribe

To subscribe to one or more of the log streams available in the Lambda execution environment, extensions send a Subscribe API request.

Path – /logs

Method – PUT

Body parameters

destination – See the section called “Destination protocols”. Required: yes. Type: strings.
buffering – See the section called “Buffering configuration”. Required: no. Type: strings.

types – An array of the types of logs to receive. Required: yes. Type: array of strings. Valid values: "platform", "function", "extension".

schemaVersion – Required: no. Default value: "2020-08-15". Set to "2021-03-18" for the extension to receive platform.runtimeDone messages.

Response parameters

The OpenAPI specifications for the subscription responses version 2020-08-15 are available for the HTTP and TCP protocols:

- HTTP: logs-api-http-response.zip
- TCP: logs-api-tcp-response.zip

Response codes

- 200 – Request completed successfully
- 202 – Request accepted. Response to a subscription request during local testing.
- 4XX – Bad Request
- 500 – Service error

If the request succeeds, the subscriber receives an HTTP 200 success response.

```
HTTP/1.1 200 OK
"OK"
```

If the request fails, the subscriber receives an error response. For example:

```
HTTP/1.1 400 OK
{
   "errorType": "Logs.ValidationError",
   "errorMessage": "URI port is not provided; types should not be empty"
}
```

Log messages

The Logs API allows extensions to subscribe to three different logs streams:
• Function – Logs that the Lambda function generates and writes to stdout or stderr.
• Extension – Logs that extension code generates.
• Platform – Logs that the runtime platform generates, which record events and errors related to
  invocations and extensions.

Topics

• Function logs
• Extension logs
• Platform logs

Function logs

The Lambda function and internal extensions generate function logs and write them to stdout or
stderr.

The following example shows the format of a function log message. { "time": "2020-08-20T12:31:32.123Z", "type": "function", "record": "ERROR encountered. Stack trace:\nmy-function (line 10)\n" }

Extension logs

Extensions can generate extension logs. The log format is the same as for a function log.

Platform logs

Lambda generates log messages for platform events such as platform.start, platform.end,
and platform.fault.

Optionally, you can subscribe to the 2021-03-18 version of the Logs API schema, which includes
the platform.runtimeDone log message.

Example platform log messages

The following example shows the platform start and platform end logs. These logs indicate the
invocation start time and invocation end time for the invocation that the requestid specifies.

{ }
The `platform.initRuntimeDone` log message shows the status of the Runtime init sub-phase, which is part of the **Init lifecycle phase**. When Runtime init is successful, the runtime sends a / next runtime API request (for the on-demand and provisioned-concurrency initialization types) or restore/next (for the snap-start initialization type). The following example shows a successful `platform.initRuntimeDone` log message for the snap-start initialization type.

```json
{
    "time":"2022-07-17T18:41:57.083Z",
    "type":"platform.initRuntimeDone",
    "record":{
        "initializationType":"snap-start",
        "status":"success"
    }
}
```

The `platform.initReport` log message shows how long the Init phase lasted and how many milliseconds you were billed for during this phase. When the initialization type is provisioned-concurrency, Lambda sends this message during invocation. When the initialization type is snap-start, Lambda sends this message after restoring the snapshot. The following example shows a `platform.initReport` log message for the snap-start initialization type.

```json
{
    "time":"2022-07-17T18:41:57.083Z",
    "type":"platform.initReport",
    "record":{
        "initializationType":"snap-start",
        "metrics":{
            "durationMs":731.79,
            "billedDurationMs":732
        }
    }
}
```
The platform report log includes metrics about the invocation that the requestId specifies. The initDurationMs field is included in the log only if the invocation included a cold start. If AWS X-Ray tracing is active, the log includes X-Ray metadata. The following example shows a platform report log for an invocation that included a cold start.

```json
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.report",
    "record": {
        "requestId": "6f7f0961f83442118a7af6fe80b88d56",
        "metrics": {
            "durationMs": 101.51,
            "billedDurationMs": 300,
            "memorySizeMB": 512,
            "maxMemoryUsedMB": 33,
            "initDurationMs": 116.67
        }
    }
}
```

The platform fault log captures runtime or execution environment errors. The following example shows a platform fault log message.

```json
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.fault",
    "record": "RequestId: d783b35e-a91d-4251-af17-035953428a2c Process exited before completing request"
}
```

Lambda generates a platform extension log when an extension registers with the extensions API. The following example shows a platform extension message.

```json
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.extension",
    "record": {
        "name": "Foo.bar",
        "state": "Ready",
        "events": ["INVOKED", "SHUTDOWN"]
    }
}
```
Lambda generates a platform logs subscription log when an extension subscribes to the logs API. The following example shows a logs subscription message.

```json
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsSubscription",
    "record": {
        "name": "Foo.bar",
        "state": "Subscribed",
        "types": ["function", "platform"
    }
}
```

Lambda generates a platform logs dropped log when an extension is not able to process the number of logs that it is receiving. The following example shows a `platform.logsDropped` log message.

```json
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsDropped",
    "record": {
        "reason": "Consumer seems to have fallen behind as it has not acknowledged receipt of logs."
        "droppedRecords": 123,
        "droppedBytes": 12345
    }
}
```

The `platform.restoreStart` log message shows the time that the Restore phase started (snap-start initialization type only). Example:

```json
{
    "time": "2022-07-17T18:43:44.782Z",
    "type": "platform.restoreStart",
    "record": {}
}
```

The `platform.restoreReport` log message shows how long the Restore phase lasted and how many milliseconds you were billed for during this phase (snap-start initialization type only). Example:

```json
{
}
```
Platform runtimeDone messages

If you set the schema version to "2021-03-18" in the subscribe request, Lambda sends a `platform.runtimeDone` message after the function invocation completes either successfully or with an error. The extension can use this message to stop all the telemetry collection for this function invocation.

The OpenAPI specification for the Log event type in schema version **2021-03-18** is available here: [schema-2021-03-18.zip](schema-2021-03-18.zip)

Lambda generates the `platform.runtimeDone` log message when the runtime sends a `Next` or `Error` runtime API request. The `platform.runtimeDone` log informs consumers of the Logs API that the function invocation completes. Extensions can use this information to decide when to send all the telemetry collected during that invocation.

**Examples**

Lambda sends the `platform.runtimeDone` message after the runtime sends the `NEXT` request when the function invocation completes. The following examples show messages for each of the status values: success, failure, and timeout.

**Example Example success message**

```json
{
    "time": "2021-02-04T20:00:05.123Z",
    "type": "platform.runtimeDone",
    "record": {
        "requestId": "6f7f0961f83442118a7af6fe80b88",
        "status": "success"
    }
}
```
Example Example failure message

```
{
    "time": "2021-02-04T20:00:05.123Z",
    "type": "platform.runtimeDone",
    "record": {
        "requestId": "6f7f0961f83442118a7af6fe80b88",
        "status": "failure"
    }
}
```

Example Example timeout message

```
{
    "time": "2021-02-04T20:00:05.123Z",
    "type": "platform.runtimeDone",
    "record": {
        "requestId": "6f7f0961f83442118a7af6fe80b88",
        "status": "timeout"
    }
}
```

Example Example platform.restoreRuntimeDone message (snap-start initialization type only)

The `platform.restoreRuntimeDone` log message shows whether or not the Restore phase was successful. Lambda sends this message when the runtime sends a restore/next runtime API request. There are three possible statuses: success, failure, and timeout. The following example shows a successful `platform.restoreRuntimeDone` log message.

```
{
    "time": "2022-07-17T18:43:45.936Z",
    "type": "platform.restoreRuntimeDone",
    "record": {
        "status": "success"
    }
}
```
Troubleshooting issues in Lambda

The following topics provide troubleshooting advice for errors and issues that you might encounter when using the Lambda API, console, or tools. If you find an issue that is not listed here, you can use the Feedback button on this page to report it.

For more troubleshooting advice and answers to common support questions, visit the AWS Knowledge Center.

For more information about debugging and troubleshooting Lambda applications, see Debugging in Serverless Land.

Topics

- Troubleshoot deployment issues in Lambda
- Troubleshoot invocation issues in Lambda
- Troubleshoot execution issues in Lambda
- Troubleshoot networking issues in Lambda
- Troubleshoot container image issues in Lambda

Troubleshoot deployment issues in Lambda

When you update your function, Lambda deploys the change by launching new instances of the function with the updated code or settings. Deployment errors prevent the new version from being used and can arise from issues with your deployment package, code, permissions, or tools.

When you deploy updates to your function directly with the Lambda API or with a client such as the AWS CLI, you can see errors from Lambda directly in the output. If you use services like AWS CloudFormation, AWS CodeDeploy, or AWS CodePipeline, look for the response from Lambda in the logs or event stream for that service.

The following topics provide troubleshooting advice for errors and issues that you might encounter when using the Lambda API, console, or tools. If you find an issue that is not listed here, you can use the Feedback button on this page to report it.

For more troubleshooting advice and answers to common support questions, visit the AWS Knowledge Center.
For more information about debugging and troubleshooting Lambda applications, see Debugging in Serverless Land.

Topics

- **General: Permission is denied / Cannot load such file**
- **General: Error occurs when calling the UpdateFunctionCode**
- **Amazon S3: Error Code PermanentRedirect.**
- **General: Cannot find, cannot load, unable to import, class not found, no such file or directory**
- **General: Undefined method handler**
- **Lambda: Layer conversion failed**
- **Lambda: InvalidParameterValueException or RequestEntityTooLargeException**
- **Lambda: InvalidParameterValueException**
- **Lambda: Concurrency and memory quotas**

**General: Permission is denied / Cannot load such file**

Error: *EACCES: permission denied, open '/var/task/index.js'*

Error: *cannot load such file -- function*

Error: *Errno 13] Permission denied: '/var/task/function.py'*

The Lambda runtime needs permission to read the files in your deployment package. In Linux permissions octal notation, Lambda needs 644 permissions for non-executable files (rw-r--r--) and 755 permissions (rwxr-xr-x) for directories and executable files.

In Linux and MacOS, use the `chmod` command to change file permissions on files and directories in your deployment package. For example, to give an executable file the correct permissions, run the following command.

```
chmod 755 <filepath>
```

To change file permissions in Windows, see Set, View, Change, or Remove Permissions on an Object in the Microsoft Windows documentation.
General: Error occurs when calling the UpdateFunctionCode

**Error:** An error occurred (RequestEntityTooLargeException) when calling the UpdateFunctionCode operation

When you upload a deployment package or layer archive directly to Lambda, the size of the ZIP file is limited to 50 MB. To upload a larger file, store it in Amazon S3 and use the S3Bucket and S3Key parameters.

**Note**

When you upload a file directly with the AWS CLI, AWS SDK, or otherwise, the binary ZIP file is converted to base64, which increases its size by about 30%. To allow for this, and the size of other parameters in the request, the actual request size limit that Lambda applies is larger. Due to this, the 50 MB limit is approximate.

Amazon S3: Error Code PermanentRedirect.

**Error:** Error occurred while GetObject. S3 Error Code: PermanentRedirect. S3 Error Message: The bucket is in this region: us-east-2. Please use this region to retry the request

When you upload a function's deployment package from an Amazon S3 bucket, the bucket must be in the same Region as the function. This issue can occur when you specify an Amazon S3 object in a call to UpdateFunctionCode, or use the package and deploy commands in the AWS CLI or AWS SAM CLI. Create a deployment artifact bucket for each Region where you develop applications.

General: Cannot find, cannot load, unable to import, class not found, no such file or directory

**Error:** Cannot find module 'function'

**Error:** cannot load such file -- function

**Error:** Unable to import module 'function'

**Error:** Class not found: function.Handler

**Error:** fork/exec /var/task/function: no such file or directory
Error: Unable to load type 'Function.Handler' from assembly 'Function'.

The name of the file or class in your function's handler configuration doesn't match your code. See the following section for more information.

**General: Undefined method handler**

**Error: index.handler is undefined or not exported**

**Error: Handler 'handler' missing on module 'function'**

**Error: undefined method `handler' for #<LambdaHandler:0x000055b76ccebf98>**

**Error: No public method named handleRequest with appropriate method signature found on class function.Handler**

**Error: Unable to find method 'handleRequest' in type 'Function.Handler' from assembly 'Function'**

The name of the handler method in your function's handler configuration doesn't match your code. Each runtime defines a naming convention for handlers, such as `filename.methodname`. The handler is the method in your function's code that the runtime runs when your function is invoked.

For some languages, Lambda provides a library with an interface that expects a handler method to have a specific name. For details about handler naming for each language, see the following topics.

- Building Lambda functions with Node.js
- Building Lambda functions with Python
- Building Lambda functions with Ruby
- Building Lambda functions with Java
- Building Lambda functions with Go
- Building Lambda functions with C#
- Building Lambda functions with PowerShell

**Lambda: Layer conversion failed**

**Error: Lambda layer conversion failed. For advice on resolving this issue, see the Troubleshoot deployment issues in Lambda page in the Lambda User Guide.**
When you configure a Lambda function with a layer, Lambda merges the layer with your function code. If this process fails to complete, Lambda returns this error. If you encounter this error, take the following steps:

- Delete any unused files from your layer
- Delete any symbolic links in your layer
- Rename any files that have the same name as a directory in any of your function's layers

Lambda: InvalidParameterValueException or RequestEntityTooLargeException

Error: InvalidParameterValueException: Lambda was unable to configure your environment variables because the environment variables you have provided exceeded the 4KB limit. String measured: 
{"A1":"uSFeY5cyPiPn7AtnX5BsM..."

Error: RequestEntityTooLargeException: Request must be smaller than 5120 bytes for the UpdateFunctionConfiguration operation

The maximum size of the variables object that is stored in the function's configuration must not exceed 4096 bytes. This includes key names, values, quotes, commas, and brackets. The total size of the HTTP request body is also limited.

```
{
    "FunctionName": "my-function",
    "Runtime": "nodejs20.x",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "my-bucket",
            "KEY": "file.txt"
        }
    }
}
```

In this example, the object is 39 characters and takes up 39 bytes when it's stored (without white space) as the string {"BUCKET":"my-bucket","KEY":"file.txt"}. Standard ASCII characters
in environment variable values use one byte each. Extended ASCII and Unicode characters can use between 2 bytes and 4 bytes per character.

**Lambda: InvalidParameterValueException**

**Error:** `InvalidParameterValueException`: Lambda was unable to configure your environment variables because the environment variables you have provided contains reserved keys that are currently not supported for modification.

Lambda reserves some environment variable keys for internal use. For example, `AWS_REGION` is used by the runtime to determine the current Region and cannot be overridden. Other variables, like `PATH`, are used by the runtime but can be extended in your function configuration. For a full list, see [Defined runtime environment variables](#).

**Lambda: Concurrency and memory quotas**

**Error:** `Specified ConcurrentExecutions for function decreases account's UnreservedConcurrentExecution below its minimum value`

**Error:** `'MemorySize' value failed to satisfy constraint: Member must have value less than or equal to 3008`

These errors occur when you exceed the concurrency or memory quotas for your account. New AWS accounts have reduced concurrency and memory quotas. To resolve errors related to concurrency, you can [request a quota increase](#). You cannot request memory quota increases.

- **Concurrency:** You might get an error if you try to create a function using reserved or provisioned concurrency, or if your per-function concurrency request ([PutFunctionConcurrency](#)) exceeds your account's concurrency quota.
- **Memory:** Errors occur if the amount of memory allocated to the function exceeds your account's memory quota.

**Troubleshoot invocation issues in Lambda**

When you invoke a Lambda function, Lambda validates the request and checks for scaling capacity before sending the event to your function or, for asynchronous invocation, to the event queue. Invocation errors can be caused by issues with request parameters, event structure, function settings, user permissions, resource permissions, or limits.
If you invoke your function directly, you see any invocation errors in the response from Lambda. If you invoke your function asynchronously with an event source mapping or through another service, you might find errors in logs, a dead-letter queue, or a failed-event destination. Error handling options and retry behavior vary depending on how you invoke your function and on the type of error.

For a list of error types that the Invoke operation can return, see Invoke.

IAM: lambda:InvokeFunction not authorized


Your user, or the role that you assume, must have permission to invoke a function. This requirement also applies to Lambda functions and other compute resources that invoke functions. Add the AWS managed policy AWSLambdaRole to your user, or add a custom policy that allows the lambda:InvokeFunction action on the target function.

Note

The name of the IAM action (lambda:InvokeFunction) refers to the Invoke Lambda API operation.

For more information, see Lambda resource access permissions.

Lambda: Couldn't find valid bootstrap (Runtime.InvalidEntrypoint)

Error: Couldn't find valid bootstrap(s): [/var/task/bootstrap /opt/bootstrap]

This error typically occurs when the root of your deployment package doesn't contain an executable file named bootstrap. For example, if you're deploying a provided.al2023 function with a .zip file, the bootstrap file must be at the root of the .zip file, not in a directory.

Lambda: Operation cannot be performed ResourceConflictException

Error: ResourceConflictException: The operation cannot be performed at this time. The function is currently in the following state: Pending

When you connect a function to a virtual private cloud (VPC) at the time of creation, the function enters a Pending state while Lambda creates elastic network interfaces. During this time, you
can't invoke or modify your function. If you connect your function to a VPC after creation, you can invoke it while the update is pending, but you can't modify its code or configuration.

For more information, see [Lambda function states](#).

**Lambda: Function is stuck in Pending**

**Error:** A function is stuck in the Pending state for several minutes.

If a function is stuck in the Pending state for more than six minutes, call one of the following API operations to unblock it:

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

Lambda cancels the pending operation and puts the function into the Failed state. You can then attempt another update.

**Lambda: One function is using all concurrency**

**Issue:** One function is using all of the available concurrency, causing other functions to be throttled.

To divide your AWS account's available concurrency in an AWS Region into pools, use reserved concurrency. Reserved concurrency ensures that a function can always scale to its assigned concurrency, and that it doesn't scale beyond its assigned concurrency.

**General: Cannot invoke function with other accounts or services**

**Issue:** You can invoke your function directly, but it doesn't run when another service or account invokes it.

You grant other services and accounts permission to invoke a function in the function's resource-based policy. If the invoker is in another account, that user must also have permission to invoke functions.

**General: Function invocation is looping**

**Issue:** Function is invoked continuously in a loop.
This typically occurs when your function manages resources in the same AWS service that triggers it. For example, it's possible to create a function that stores an object in an Amazon Simple Storage Service (Amazon S3) bucket that's configured with a notification that invokes the function again. To stop the function from running, on the function configuration page, choose Throttle. Then, identify the code path or configuration error that caused the recursive invocation. Lambda automatically detects and stops recursive loops for some AWS services and SDKs. See the section called “Recursive loop detection” for more details.

Lambda: Alias routing with provisioned concurrency

Issue: Provisioned concurrency spillover invocations during alias routing.

Lambda uses a simple probabilistic model to distribute the traffic between the two function versions. At low traffic levels, you might see a high variance between the configured and actual percentage of traffic on each version. If your function uses provisioned concurrency, you can avoid spillover invocations by configuring a higher number of provisioned concurrency instances during the time that alias routing is active.

Lambda: Cold starts with provisioned concurrency

Issue: You see cold starts after enabling provisioned concurrency.

When the number of concurrent executions on a function is less than or equal to the configured level of provisioned concurrency, there shouldn't be any cold starts. To help you confirm if provisioned concurrency is operating normally, do the following:

- Check that provisioned concurrency is enabled on the function version or alias.

  Note
  Provisioned concurrency is not configurable on the unpublished version of the function ($LATEST).

- Ensure that your triggers invoke the correct function version or alias. For example, if you're using Amazon API Gateway, check that API Gateway invokes the function version or alias with provisioned concurrency, not $LATEST. To confirm that provisioned concurrency is being used, you can check the ProvisionedConcurrencyInvocations Amazon CloudWatch metric. A non-zero value indicates that the function is processing invocations on initialized execution environments.
• Determine whether your function concurrency exceeds the configured level of provisioned concurrency by checking the [ProvisionedConcurrencySpilloverInvocations CloudWatch metric](https://docs.aws.amazon.com/lambda/latest/dg/concurrency-considerations.html). A non-zero value indicates that all provisioned concurrency is in use and some invocation occurred with a cold start.

• Check your [invocation frequency](https://docs.aws.amazon.com/lambda/latest/dg/concurrency-considerations.html) (requests per second). Functions with provisioned concurrency have a maximum rate of 10 requests per second per provisioned concurrency. For example, a function configured with 100 provisioned concurrency can handle 1,000 requests per second. If the invocation rate exceeds 1,000 requests per second, some cold starts can occur.

**Lambda: Cold starts with new versions**

**Issue:** You see cold starts while deploying new versions of your function.

When you update a function alias, Lambda automatically shifts provisioned concurrency to the new version based on the weights configured on the alias.

**Error:** `KMSDisabledException`: Lambda was unable to decrypt the environment variables because the KMS key used is disabled. Please check the function's KMS key settings.

This error can occur if your AWS Key Management Service (AWS KMS) key is disabled, or if the grant that allows Lambda to use the key is revoked. If the grant is missing, configure the function to use a different key. Then, reassign the custom key to recreate the grant.

**EFS: Function could not mount the EFS file system**

**Error:** `EFSMountFailureException`: The function could not mount the EFS file system with access point arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd.

The mount request to the function's file system was rejected. Check the function's permissions, and confirm that its file system and access point exist and are ready for use.

**EFS: Function could not connect to the EFS file system**

**Error:** `EFSMountConnectivityException`: The function couldn't connect to the Amazon EFS file system with access point arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd. Check your network configuration and try again.

The function couldn't establish a connection to the function's file system with the NFS protocol (TCP port 2049). Check the security group and routing configuration for the VPC's subnets.
EFS: Function could not mount the EFS file system due to timeout

**Error:** `EFSMountTimeoutException`: The function could not mount the EFS file system with access point `{arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd}` due to mount time out.

The function could connect to the function's file system, but the mount operation timed out. Try again after a short time and consider limiting the function's concurrency to reduce load on the file system.

**Lambda: Lambda detected an IO process that was taking too long**

`EFSIOException`: This function instance was stopped because Lambda detected an IO process that was taking too long.

A previous invocation timed out and Lambda couldn't terminate the function handler. This issue can occur when an attached file system runs out of burst credits and the baseline throughput is insufficient. To increase throughput, you can increase the size of the file system or use provisioned throughput. For more information, see [Throughput](#).

**Troubleshoot execution issues in Lambda**

When the Lambda runtime runs your function code, the event might be processed on an instance of the function that's been processing events for some time, or it might require a new instance to be initialized. Errors can occur during function initialization, when your handler code processes the event, or when your function returns (or fails to return) a response.

Function execution errors can be caused by issues with your code, function configuration, downstream resources, or permissions. If you invoke your function directly, you see function errors in the response from Lambda. If you invoke your function asynchronously, with an event source mapping, or through another service, you might find errors in logs, a dead-letter queue, or an on-failure destination. Error handling options and retry behavior vary depending on how you invoke your function and on the type of error.

When your function code or the Lambda runtime return an error, the status code in the response from Lambda is 200 OK. The presence of an error in the response is indicated by a header named `X-Amz-Function-Error`. 400 and 500-series status codes are reserved for invocation errors.
Lambda: Execution takes too long

**Issue:** Function execution takes too long.

If your code takes much longer to run in Lambda than on your local machine, it may be constrained by the memory or processing power available to the function. Configure the function with additional memory to increase both memory and CPU.

Lambda: Logs or traces don't appear

**Issue:** Logs don't appear in CloudWatch Logs.

**Issue:** Traces don't appear in AWS X-Ray.

Your function needs permission to call CloudWatch Logs and X-Ray. Update its execution role to grant it permission. Add the following managed policies to enable logs and tracing.

- AWSLambdaBasicExecutionRole
- AWSXRayDaemonWriteAccess

When you add permissions to your function, update its code or configuration as well. This forces running instances of your function, which have outdated credentials, to stop and be replaced.

Note

It may take 5 to 10 minutes for logs to show up after a function invocation.

Lambda: Not all of my function's logs appear

**Issue:** Function logs are missing in CloudWatch Logs, even though my permissions are correct

If your AWS account reaches its CloudWatch Logs quota limits, CloudWatch throttles function logging. When this happens, some of the logs output by your functions may not appear in CloudWatch Logs.

If your function outputs logs at too high a rate for Lambda to process them, this can also cause log outputs not to appear in CloudWatch Logs. When Lambda can't send logs to CloudWatch at the rate your function produces them, it drops logs to prevent the execution of your function from slowing down.
To check if your AWS account has reached its CloudWatch Logs quota limits, do the following:

1. Open the Service Quotas console.
2. In the navigation pane, choose AWS services.
3. From the AWS services list, search for Amazon CloudWatch Logs.
4. In the Service quotas list, choose the CreateLogGroup throttle limit in transactions per second, CreateLogStream throttle limit in transactions per second and PutLogEvents throttle limit in transactions per second quotas to view your utilization.

You can also set CloudWatch alarms to alert you when your account utilization exceeds a limit you specify for these quotas. See Create a CloudWatch alarm based on a static threshold to learn more.

If the default quota limits for CloudWatch Logs aren't enough for your use case, you can request a quota increase.

**Lambda: The function returns before execution finishes**

**Issue: (Node.js) Function returns before code finishes executing**

Many libraries, including the AWS SDK, operate asynchronously. When you make a network call or perform another operation that requires waiting for a response, libraries return an object called a promise that tracks the progress of the operation in the background.

To wait for the promise to resolve into a response, use the await keyword. This blocks your handler code from executing until the promise is resolved into an object that contains the response. If you don't need to use the data from the response in your code, you can return the promise directly to the runtime.

Some libraries don't return promises but can be wrapped in code that does. For more information, see AWS Lambda function handler in Node.js.

**AWS SDK: Versions and updates**

**Issue: The AWS SDK included on the runtime is not the latest version**

**Issue: The AWS SDK included on the runtime updates automatically**

Runtimes for scripting languages include the AWS SDK and are periodically updated to the latest version. The current version for each runtime is listed on runtimes page. To use a newer version
of the AWS SDK, or to lock your functions to a specific version, you can bundle the library with
your function code, or create a Lambda layer. For details on creating a deployment package with
dependencies, see the following topics:

Node.js

Deploy Node.js Lambda functions with .zip file archives

Python

Working with .zip file archives for Python Lambda functions

Ruby

Working with .zip file archives for Ruby Lambda functions

Java

Deploy Java Lambda functions with .zip or JAR file archives

Go

Deploy Go Lambda functions with .zip file archives

C#

Build and deploy C# Lambda functions with .zip file archives

PowerShell

Deploy PowerShell Lambda functions with .zip file archives

Python: Libraries load incorrectly

Issue: (Python) Some libraries don't load correctly from the deployment package

Libraries with extension modules written in C or C++ must be compiled in an environment with
the same processor architecture as Lambda (Amazon Linux). For more information, see Working
with .zip file archives for Python Lambda functions.

Troubleshoot networking issues in Lambda

By default, Lambda runs your functions in an internal virtual private cloud (VPC) with connectivity
to AWS services and the internet. To access local network resources, you can configure your
function to connect to a VPC in your account. When you use this feature, you manage the
function's internet access and network connectivity with Amazon Virtual Private Cloud (Amazon VPC) resources.

Network connectivity errors can result from issues with your VPC's routing configuration, security group rules, AWS Identity and Access Management (IAM) role permissions, or network address translation (NAT), or from the availability of resources such as IP addresses or network interfaces. Depending on the issue, you might see a specific error or timeout if a request can't reach its destination.

**VPC: Function loses internet access or times out**

**Issue:** Your Lambda function loses internet access after connecting to a VPC.

**Error:** Error: connect ETIMEDOUT 176.32.98.189:443

**Error:** Error: Task timed out after 10.00 seconds

**Error:** ReadTimeoutError: Read timed out. (read timeout=15)

When you connect a function to a VPC, all outbound requests go through the VPC. To connect to the internet, configure your VPC to send outbound traffic from the function's subnet to a NAT gateway in a public subnet. For more information and sample VPC configurations, see Internet and service access for VPC-connected functions.

If some of your TCP connections are timing out, this may be due to packet fragmentation. Lambda functions cannot handle incoming fragmented TCP requests, since Lambda does not support IP fragmentation for TCP or ICMP.

**VPC: Function needs access to AWS services without using the internet**

**Issue:** Your Lambda function needs access to AWS services without using the internet.

To connect a function to AWS services from a private subnet with no internet access, use VPC endpoints. For a sample AWS CloudFormation template with VPC endpoints for Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB (DynamoDB), see Sample VPC configurations.

**VPC: Elastic network interface limit reached**

**Error:** ENILimitReachedException: The elastic network interface limit was reached for the function's VPC.
When you connect a Lambda function to a VPC, Lambda creates an elastic network interface for each combination of subnet and security group attached to the function. The default service quota is 250 network interfaces per VPC. To request a quota increase, use the Service Quotas console.

**EC2: Elastic network interface with type of "lambda"**

**Error Code:** `Client.OperationNotPermitted`

**Error message:** *The security group can not be modified for this type of interface*

You will receive this error if you attempt to modify an elastic network interface (ENI) that is managed by Lambda. The `ModifyNetworkInterfaceAttribute` is not included in the Lambda API for update operations on elastic network interfaces created by Lambda.

**Troubleshoot container image issues in Lambda**

**Container: CodeArtifactUserException errors related to the code artifact.**

**Issue:** `CodeArtifactUserPendingException error message`

The CodeArtifact is pending optimization. The function transitions to active state when Lambda completes the optimization. HTTP response code 409.

**Issue:** `CodeArtifactUserDeletedException error message`

The CodeArtifact is scheduled to be deleted. HTTP response code 409.

**Issue:** `CodeArtifactUserFailedException error message`

Lambda failed to optimize the code. You need to correct the code and upload it again. HTTP response code 409.

**Container: ManifestKeyCustomerException errors related to the code manifest key.**

**Issue:** `KMSAccessDeniedException error message`

You do not have permissions to access the key to decrypt the manifest. HTTP response code 502.
**Issue: TooManyRequestsException error message**

The client is being throttled. The current request rate exceeds the KMS subscription rate. HTTP response code 429.

**Issue: KMSNotFoundException error message**

Lambda cannot find the key to decrypt the manifest. HTTP response code 502.

**Issue: KMSDisabledException error message**

The key to decrypt the manifest is disabled. HTTP response code 502.

**Issue: KMSInvalidStateException error message**

The key is in a state (such as pending deletion or unavailable) such that Lambda cannot use the key to decrypt the manifest. HTTP response code 502.

**Container: Error occurs on runtime InvalidEntrypoint**

**Issue: You receive a Runtime.ExitError error message, or an error message with "errorType": "Runtime.InvalidEntrypoint".**

Verify that the ENTRYPONT to your container image includes the absolute path as the location. Also verify that the image does not contain a symlink as the ENTRYPONT.

**Lambda: System provisioning additional capacity**

**Error: “Error: We currently do not have sufficient capacity in the region you requested. Our system will be working on provisioning additional capacity.**

Retry the function invocation. If the retry fails, validate that the files required to run the function code can be read by any user. Lambda defines a default Linux user with least-privileged permissions. You need to verify that your application code does not rely on files that are restricted by other Linux users for execution.

**CloudFormation: ENTRYPONT is being overridden with a null or empty value**

**Error: You are using an AWS CloudFormation template, and your container ENTRYPONT is being overridden with a null or empty value.**
Review the ImageConfig resource in the AWS CloudFormation template. If you declare an ImageConfig resource in your template, you must provide non-empty values for all three of the properties.
AWS Lambda applications

An AWS Lambda application is a combination of Lambda functions, event sources, and other resources that work together to perform tasks. You can use AWS CloudFormation and other tools to collect your application's components into a single package that can be deployed and managed as one resource. Applications make your Lambda projects portable and enable you to integrate with additional developer tools, such as AWS CodePipeline, AWS CodeBuild, and the AWS Serverless Application Model command line interface (AWS SAM CLI).

The AWS Serverless Application Repository provides a collection of Lambda applications that you can deploy in your account with a few clicks. The repository includes both ready-to-use applications and samples that you can use as a starting point for your own projects. You can also submit your own projects for inclusion.

AWS CloudFormation enables you to create a template that defines your application's resources and lets you manage the application as a stack. You can more safely add or modify resources in your application stack. If any part of an update fails, AWS CloudFormation automatically rolls back to the previous configuration. With AWS CloudFormation parameters, you can create multiple environments for your application from the same template. AWS SAM extends AWS CloudFormation with a simplified syntax focused on Lambda application development.

The AWS CLI and AWS SAM CLI are command line tools for managing Lambda application stacks. In addition to commands for managing application stacks with the AWS CloudFormation API, the AWS CLI supports higher-level commands that simplify tasks such as uploading deployment packages and updating templates. The AWS SAM CLI provides additional functionality, including validating templates, testing locally, and integrating with CI/CD systems.

When creating an application, you can create its Git repository using either CodeCommit or an AWS CodeStar connection to GitHub. CodeCommit enables you to use the IAM console to manage SSH keys and HTTP credentials for your users. AWS CodeStar Connections enables you to connect to your GitHub account. For more information about connections, see What are connections? in the Developer Tools console User Guide.

For more information about designing Lambda applications, see Application design in Serverless Land.

Topics

- Managing applications in the AWS Lambda console
• Creating an application with continuous delivery in the Lambda console
• Rolling deployments for Lambda functions
• Invoking Lambda functions with the AWS Mobile SDK for Android
Managing applications in the AWS Lambda console

The AWS Lambda console helps you monitor and manage your Lambda applications. The Applications menu lists AWS CloudFormation stacks with Lambda functions. The menu includes stacks that you launch in AWS CloudFormation by using the AWS CloudFormation console, the AWS Serverless Application Repository, the AWS CLI, or the AWS SAM CLI.

To view a Lambda application

1. Open the Lambda console Applications page.
2. Choose an application.

The overview shows the following information about your application.

- AWS CloudFormation template or SAM template – The template that defines your application.
- Resources – The AWS resources that are defined in your application's template. To manage your application's Lambda functions, choose a function name from the list.

Monitoring applications

The Monitoring tab shows an Amazon CloudWatch dashboard with aggregate metrics for the resources in your application.

To monitor a Lambda application

1. Open the Lambda console Applications page.
2. Choose Monitoring.

By default, the Lambda console shows a basic dashboard. You can customize this page by defining custom dashboards in your application template. When your template includes one or more
dashboards, the page shows your dashboards instead of the default dashboard. You can switch between dashboards with the drop-down menu on the top right of the page.

**Custom monitoring dashboards**

Customize your application monitoring page by adding one or more Amazon CloudWatch dashboards to your application template with the `AWS::CloudWatch::Dashboard` resource type. The following example creates a dashboard with a single widget that graphs the number of invocations of a function named `my-function`.

**Example function dashboard template**

```json
Resources:
MyDashboard:
  Type: AWS::CloudWatch::Dashboard
  Properties:
    DashboardName: my-dashboard
    DashboardBody: |
      |
        "widgets": [
          |
            |
              "type": "metric",
              "width": 12,
              "height": 6,
              "properties": {
                "metrics": [
                  |
                    "AWS/Lambda",
                    "Invocations",
                    "FunctionName",
                    "my-function",
                    |
                      "stat": "Sum",
                      "label": "MyFunction"
                    |
                ],
                |
                [ |
                  |
                    |
                      "expression": "SUM(METRICS())",
                      "label": "Total Invocations"
                  |
                ]
              }
            |
          ],
        |
```
You can get the definition for any of the widgets in the default monitoring dashboard from the CloudWatch console.

To view a widget definition

1. Open the Lambda console Applications page.
2. Choose an application that has the standard dashboard.
3. Choose Monitoring.
4. On any widget, choose View in metrics from the drop-down menu.
5. Choose Source.

For more information about authoring CloudWatch dashboards and widgets, see Dashboard body structure and syntax in the Amazon CloudWatch API Reference.
Creating an application with continuous delivery in the Lambda console

You can use the Lambda console to create an application with an integrated continuous delivery pipeline. With continuous delivery, every change that you push to your source control repository triggers a pipeline that builds and deploys your application automatically. The Lambda console provides starter projects for common application types with Node.js sample code and templates that create supporting resources.

In this tutorial, you create the following resources.

- **Application** – A Node.js Lambda function, build specification, and AWS Serverless Application Model (AWS SAM) template.
- **Pipeline** – An AWS CodePipeline pipeline that connects the other resources to enable continuous delivery.
- **Repository** – A Git repository in AWS CodeCommit. When you push a change, the pipeline copies the source code into an Amazon S3 bucket and passes it to the build project.
- **Trigger** – An Amazon EventBridge (CloudWatch Events) rule that watches the main branch of the repository and triggers the pipeline.
- **Build project** – An AWS CodeBuild build that gets the source code from the pipeline and packages the application. The source includes a build specification with commands that install dependencies and prepare the application template for deployment.
- **Deployment configuration** – The pipeline's deployment stage defines a set of actions that take the processed AWS SAM template from the build output, and deploy the new version with AWS CloudFormation.
- **Bucket** – An Amazon Simple Storage Service (Amazon S3) bucket for deployment artifact storage.
- **Roles** – The pipeline's source, build, and deploy stages have IAM roles that allow them to manage AWS resources. The application's function has an execution role that allows it to upload logs and can be extended to access other services.

Your application and pipeline resources are defined in AWS CloudFormation templates that you can customize and extend. Your application repository includes a template that you can modify to add Amazon DynamoDB tables, an Amazon API Gateway API, and other application resources. The
continuous delivery pipeline is defined in a separate template outside of source control and has its own stack.

The pipeline maps a single branch in a repository to a single application stack. You can create additional pipelines to add environments for other branches in the same repository. You can also add stages to your pipeline for testing, staging, and manual approvals. For more information about AWS CodePipeline, see What is AWS CodePipeline.

Sections

- Prerequisites
- Create an application
- Invoke the function
- Add an AWS resource
- Update the permissions boundary
- Update the function code
- Next steps
- Troubleshooting
- Clean up

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in Create a Lambda function with the console to create your first Lambda function.

To complete the following steps, you need the AWS Command Line Interface (AWS CLI) version 2. Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.
Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux. Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

This tutorial uses CodeCommit for source control. To set up your local machine to access and update application code, see Setting up in the AWS CodeCommit User Guide.

Create an application

Create an application in the Lambda console. In Lambda, an application is an AWS CloudFormation stack with a Lambda function and any number of supporting resources. In this tutorial, you create an application that has a function and its execution role.

To create an application

1. Open the Lambda console Applications page.
2. Choose Create application.
3. Choose Author from scratch.
4. Configure application settings.
   - Name – my-app.
   - Runtime – Node.js 18.x.
   - Template format – AWS SAM (YAML).
   - Source control service – CodeCommit.
   - Repository name – my-app-repo.
   - Permissions – Create roles and permissions boundary.
5. Choose Create.

Lambda creates the pipeline and related resources and commits the sample application code to the Git repository. As resources are created, they appear on the overview page.
The **Infrastructure** stack contains the repository, build project, and other resources that combine to form a continuous delivery pipeline. When this stack finishes deploying, it in turn deploys the application stack that contains the function and execution role. These are the application resources that appear under **Resources**.

**Invoke the function**

When the deployment process completes, invoke the function from the Lambda console.

**To invoke the application's function**

1. Open the Lambda console [Applications page].
2. Choose **my-app**.
3. Under **Resources**, choose **helloFromLambdaFunction**.
4. Choose **Test**.
5. Configure a test event.
   - **Event name** – event
   - **Body** – {}
6. Choose **Create**.

7. Choose **Test**.

The Lambda console runs your function and displays the result. Expand the **Details** section under the result to see the output and execution details.

![Execution result: succeeded (logs)](image)

**Add an AWS resource**

In the previous step, Lambda console created a Git repository that contains function code, a template, and a build specification. You can add resources to your application by modifying the template and pushing changes to the repository. To get a copy of the application on your local machine, clone the repository.

**To clone the project repository**

1. Open the Lambda console [Applications page](#).
2. Choose **my-app**.
3. Choose **Code**.
4. Under **Repository details**, copy the HTTP or SSH repository URI, depending on the authentication mode that you configured during **setup**.
5. To clone the repository, use the `git clone` command.

```
git clone ssh://git-codecommit.us-east-2.amazonaws.com/v1/repos/my-app-repo
```
To add a DynamoDB table to the application, define an AWS::Serverless::SimpleTable resource in the template.

**To add a DynamoDB table**

1. Open `template.yml` in a text editor.
2. Add a table resource, an environment variable that passes the table name to the function, and a permissions policy that allows the function to manage it.

**Example template.yml - resources**

```yaml
...
Resources:
  ddbTable:
    Type: AWS::Serverless::SimpleTable
    Properties:
      PrimaryKey:
        Name: id
        Type: String
      ProvisionedThroughput:
        ReadCapacityUnits: 1
        WriteCapacityUnits: 1
  helloFromLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./
      Handler: src/handlers/hello-from-lambda.helloFromLambdaHandler
      Runtime: nodejs18.x
      MemorySize: 128
      Timeout: 60
      Description: A Lambda function that returns a static string.
    Environment:
      Variables:
        DDB_TABLE: !Ref ddbTable
    Policies:
      - DynamoDBCrudPolicy:
          TableName: !Ref ddbTable
      - AWSLambdaBasicExecutionRole
```

3. Commit and push the change.

```
git commit -am "Add DynamoDB table"
```
When you push a change, it triggers the application's pipeline. Use the **Deploymen****ts** tab of the application screen to track the change as it flows through the pipeline. When the deployment is complete, proceed to the next step.

### Update the permissions boundary

The sample application applies a *permissions boundary* to its function's execution role. The permissions boundary limits the permissions that you can add to the function's role. Without the boundary, users with write access to the project repository could modify the project template to give the function permission to access resources and services outside of the scope of the sample application.

In order for the function to use the DynamoDB permission that you added to its execution role in the previous step, you must extend the permissions boundary to allow the additional permissions. The Lambda console detects resources that aren't in the permissions boundary and provides an updated policy that you can use to update it.
To update the application's permissions boundary

1. Open the Lambda console [Applications page].
2. Choose your application.
4. Follow the instructions shown to update the boundary to allow access to the new table.

For more information about permissions boundaries, see [Using permissions boundaries for AWS Lambda applications](#).

**Update the function code**

Next, update the function code to use the table. The following code uses the DynamoDB table to track the number of invocations processed by each instance of the function. It uses the log stream ID as a unique identifier for the function instance.

**To update the function code**

1. Add a new handler named index.js to the src/handlers folder with the following content.

   **Example src/handlers/index.js**

```javascript
const dynamodb = require('aws-sdk/clients/dynamodb');
const docClient = new dynamodb.DocumentClient();

exports.handler = async (event, context) => {
    const message = 'Hello from Lambda!';
    const tableName = process.env.DDB_TABLE;
    const logStreamName = context.logStreamName;
    var params = {
        TableName : tableName,
        Key: { id : logStreamName },
        UpdateExpression: 'set invocations = if_not_exists(invocations, :start) + :inc',
        ExpressionAttributeValues: {
            ':start': 0,
            ':inc': 1
        },
        ReturnValues: 'ALL_NEW'
    };
    await docClient.update(params).promise();
```

Update the function code
const response = {
    body: JSON.stringify(message)
};
console.log(`body: ${response.body}`);
return response;

2. Open the application template and change the handler value to `src/handlers/index.handler`.

Example template.yml

```yaml
... helloFromLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./
      Handler: src/handlers/index.handler
      Runtime: nodejs8.x
```

3. Commit and push the change.

```bash
git add . && git commit -m "Use DynamoDB table"
git push
```

After the code change is deployed, invoke the function a few times to update the DynamoDB table.

To view the DynamoDB table

1. Open the [Tables page of the DynamoDB console](https://console.aws.amazon.com/dynamodb/

2. Choose the table that starts with `my-app`.

3. Choose **Items**.

4. Choose **Start search**.
Lambda creates additional instances of your function to handle multiple concurrent invocations. Each log stream in the CloudWatch Logs log group corresponds to a function instance. A new function instance is also created when you change your function's code or configuration. For more information on scaling, see [Lambda function scaling](#).

**Next steps**

The AWS CloudFormation template that defines your application resources uses the AWS Serverless Application Model transform to simplify the syntax for resource definitions, and automate uploading the deployment package and other artifacts. AWS SAM also provides a command line interface (the AWS SAM CLI), which has the same packaging and deployment functionality as the AWS CLI, with additional features specific to Lambda applications. Use the AWS SAM CLI to test your application locally in a Docker container that emulates the Lambda execution environment.

- [Installing the AWS SAM CLI](#)
- [Testing and debugging serverless applications with AWS SAM](#)
- [Deploying serverless applications using CI/CD systems with AWS SAM](#)

AWS Cloud9 provides an online development environment that includes Node.js, the AWS SAM CLI, and Docker. With AWS Cloud9, you can start developing quickly and access your development environment from any computer. For instructions, see [Getting started](#) in the AWS Cloud9 User Guide.

For local development, AWS toolkits for integrated development environments (IDEs) let you test and debug functions before pushing them to your repository.
AWS Toolkit for JetBrains – Plugin for PyCharm (Python) and IntelliJ (Java) IDEs.

AWS Toolkit for Eclipse – Plugin for Eclipse IDE (multiple languages).


Troubleshooting

As you develop your application, you will likely encounter the following types of errors.

- **Build errors** – Issues that occur during the build phase, including compilation, test, and packaging errors.
- **Deployment errors** – Issues that occur when AWS CloudFormation isn't able to update the application stack. These include permissions errors, account quotas, service issues, or template errors.
- **Invocation errors** – Errors that are returned by a function's code or runtime.

For build and deployment errors, you can identify the cause of an error in the Lambda console.

**To troubleshoot application errors**

1. Open the Lambda console [Applications page].
2. Choose an application.
3. Choose **Deployments**.
4. To view the application's pipeline, choose **Deployment pipeline**.
5. Identify the action that encountered an error.
6. To view the error in context, choose **Details**.

For deployment errors that occur during the **ExecuteChangeSet** action, the pipeline links to a list of stack events in the AWS CloudFormation console. Search for an event with the status **UPDATE_FAILED**. Because AWS CloudFormation rolls back after an error, the relevant event is under several other events in the list. If AWS CloudFormation could not create a change set, the error appears under **Change sets** instead of under **Events**.

A common cause of deployment and invocation errors is a lack of permissions in one or more roles. The pipeline has a role for deployments (CloudFormationRole) that's equivalent to the
**user permissions** that you would use to update an AWS CloudFormation stack directly. If you add resources to your application or enable Lambda features that require user permissions, the deployment role is used. You can find a link to the deployment role under **Infrastructure** in the application overview.

If your function accesses other AWS services or resources, or if you enable features that require the function to have additional permissions, the function's **execution role** is used. All execution roles that are created in your application template are also subject to the application's permissions boundary. This boundary requires you to explicitly grant access to additional services and resources in IAM after adding permissions to the execution role in the template.

For example, to **connect a function to a virtual private cloud** (VPC), you need user permissions to describe VPC resources. The execution role needs permission to manage network interfaces. This requires the following steps.

1. Add the required user permissions to the deployment role in IAM.
2. Add the execution role permissions to the permissions boundary in IAM.
3. Add the execution role permissions to the execution role in the application template.
4. Commit and push to deploy the updated execution role.

After you address permissions errors, choose **Release change** in the pipeline overview to rerun the build and deployment.

**Clean up**

You can continue to modify and use the sample to develop your own application. If you are done using the sample, delete the application to avoid paying for the pipeline, repository, and storage.

**To delete the application**

1. Open the **AWS CloudFormation console**.
2. Delete the application stack – **my-app**.
3. Open the **Amazon S3 console**.
4. Delete the artifact bucket – **us-east-2-123456789012-my-app-pipe**.
5. Return to the AWS CloudFormation console and delete the infrastructure stack – **serverlessrepo-my-app-toolchain**.
Function logs are not associated with the application or infrastructure stack in AWS CloudFormation. Delete the log group separately in the CloudWatch Logs console.

To delete the log group

1. Open the Log groups page of the Amazon CloudWatch console.
2. Choose the function's log group (/aws/lambda/my-app-helloFromLambdaFunction-\textit{YV1VXMPLK7QK}).
3. Choose Actions, and then choose Delete log group.
4. Choose Yes, Delete.
Rolling deployments for Lambda functions

Use rolling deployments to control the risks associated with introducing new versions of your Lambda function. In a rolling deployment, the system automatically deploys the new version of the function and gradually sends an increasing amount of traffic to the new version. The amount of traffic and rate of increase are parameters that you can configure.

You configure a rolling deployment by using AWS CodeDeploy and AWS SAM. CodeDeploy is a service that automates application deployments to Amazon computing platforms such as Amazon EC2 and AWS Lambda. For more information, see What is CodeDeploy?. By using CodeDeploy to deploy your Lambda function, you can easily monitor the status of the deployment and initiate a rollback if you detect any issues.

AWS SAM is an open-source framework for building serverless applications. You create an AWS SAM template (in YAML format) to specify the configuration of the components required for the rolling deployment. AWS SAM uses the template to create and configure the components. For more information, see What is the AWS SAM?

In a rolling deployment, AWS SAM performs these tasks:

- It configures your Lambda function and creates an alias.

  The alias routing configuration is the underlying capability that implements the rolling deployment.

- It creates a CodeDeploy application and deployment group.

  The deployment group manages the rolling deployment and the rollback (if needed).

- It detects when you create a new version of your Lambda function.

- It triggers CodeDeploy to start the deployment of the new version.

Example AWS SAM Lambda template

The following example shows an AWS SAM template for a simple rolling deployment.

```
AWSTemplateFormatVersion : '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: A sample SAM template for deploying Lambda functions.
```
Resources:
# Details about the myDateTimeFunction Lambda function
myDateTimeFunction:
  Type: AWS::Serverless::Function
  Properties:
    Handler: myDateTimeFunction.handler
    Runtime: nodejs18.x
# Creates an alias named "live" for the function, and automatically publishes when you update the function.
  AutoPublishAlias: live
  DeploymentPreference:
# Specifies the deployment configuration
  Type: Linear10PercentEvery2Minutes

This template defines a Lambda function named myDateTimeFunction with the following properties.

AutoPublishAlias

The AutoPublishAlias property creates an alias named live. In addition, the AWS SAM framework automatically detects when you save new code for the function. The framework then publishes a new function version and updates the live alias to point to the new version.

DeploymentPreference

The DeploymentPreference property determines the rate at which the CodeDeploy application shifts traffic from the original version of the Lambda function to the new version. The value Linear10PercentEvery2Minutes shifts an additional ten percent of the traffic to the new version every two minutes.

For a list of the predefined deployment configurations, see Deployment configurations.

For a detailed tutorial on how to use CodeDeploy with Lambda functions, see Deploy an updated Lambda function with CodeDeploy.
Invoking Lambda functions with the AWS Mobile SDK for Android

You can call a Lambda function from a mobile application. Put business logic in functions to separate its development lifecycle from that of front-end clients, making mobile applications less complex to develop and maintain. With the Mobile SDK for Android, you use Amazon Cognito to authenticate users and authorize requests.

When you invoke a function from a mobile application, you choose the event structure, invocation type, and permission model. You can use aliases to enable seamless updates to your function code, but otherwise the function and application are tightly coupled. As you add more functions, you can create an API layer to decouple your function code from your front-end clients and improve performance.

To create a fully-featured web API for your mobile and web applications, use Amazon API Gateway. With API Gateway, you can add custom authorizers, throttle requests, and cache results for all of your functions. For more information, see Using AWS Lambda with Amazon API Gateway.

Topics
- Tutorial: Using AWS Lambda with the Mobile SDK for Android
- Sample function code

Tutorial: Using AWS Lambda with the Mobile SDK for Android

In this tutorial, you create a simple Android mobile application that uses Amazon Cognito to get credentials and invokes a Lambda function.

The mobile application retrieves AWS credentials from an Amazon Cognito identity pool and uses them to invoke a Lambda function with an event that contains request data. The function processes the request and returns a response to the front-end.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in Create a Lambda function with the console to create your first Lambda function.
To complete the following steps, you need the AWS Command Line Interface (AWS CLI) version 2. Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

⚠️ **Note**

In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system’s built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux. Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

---

**Create the execution role**

Create the [execution role](https://docs.aws.amazon.com/lambda/latest/dg/lambda-execution-role.html) that gives your function permission to access AWS resources.

**To create an execution role**

2. Choose **Create role**.
3. Create a role with the following properties.

   • **Trusted entity** – AWS Lambda.
   • **Permissions** – `AWSLambdaBasicExecutionRole`.
   • **Role name** – `lambda-android-role`.

The `AWSLambdaBasicExecutionRole` policy has the permissions that the function needs to write logs to CloudWatch Logs.
Create the function

The following example uses data to generate a string response.

Note
For sample code in other languages, see Sample function code.

Example index.js

```javascript
exports.handler = function(event, context, callback) {
    console.log("Received event: ", event);
    var data = {
        "greetings": "Hello, " + event.firstName + " " + event.lastName + "."
    };
    callback(null, data);
}
```

To create the function

1. Copy the sample code into a file named index.js.
2. Create a deployment package.

```bash
zip function.zip index.js
```
3. Create a Lambda function with the create-function command.

```bash
aws lambda create-function --function-name AndroidBackendLambdaFunction \ --zip-file fileb://function.zip --handler index.handler --runtime nodejs16.x \ --role arn:aws:iam::111122223333:role/lambda-android-role
```

Test the Lambda function

Invoke the function manually using the sample event data.

To test the Lambda function (AWS CLI)

1. Save the following sample event JSON in a file, input.txt.
2. Run the following invoke command:

```bash
aws lambda invoke --function-name AndroidBackendLambdaFunction \
--payload file://file-path/input.txt outputfile.txt
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see the [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-commands.html) in the AWS Command Line Interface User Guide for Version 2.

### Create an Amazon Cognito identity pool

In this section, you create an Amazon Cognito identity pool. The identity pool has two IAM roles. You update the IAM role for unauthenticated users and grant permissions to run the `AndroidBackendLambdaFunction` Lambda function.

For more information about IAM roles, see the [IAM roles](https://docs.aws.amazon.com/iam/latest/UserGuide/roles.html) in the IAM User Guide. For more information about Amazon Cognito services, see the Amazon Cognito product detail page.

To create an identity pool

1. Open the [Amazon Cognito console](https://console.aws.amazon.com/cognito/).
2. Create a new identity pool called `JavaFunctionAndroidEventHandlerPool`. Before you follow the procedure to create an identity pool, note the following:

   - The identity pool you are creating must allow access to unauthenticated identities because our example mobile application does not require a user log in. Therefore, make sure to select the `Enable access to unauthenticated identities` option.
   - Add the following statement to the permission policy associated with the unauthenticated identities:

```json
{
    "Effect": "Allow",
    "Action": [ "lambda:InvokeFunction"
]
}
```
The resulting policy will be as follows:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "mobileanalytics:PutEvents",
        "cognito-sync:*"
      ],
      "Resource": [
        "*
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "lambda:invokefunction"
      ],
      "Resource": [
      ]
    }
  ]
}
```

For instructions about how to create an identity pool, log in to the Amazon Cognito console and follow the New Identity Pool wizard.

3. Note the identity pool ID. You specify this ID in your mobile application you create in the next section. The app uses this ID when it sends request to Amazon Cognito to request for temporary security credentials.
Create an Android application

Create a simple Android mobile application that generates events and invokes Lambda functions by passing the event data as parameters.

The following instructions have been verified using Android studio.

1. Create a new Android project called AndroidEventGenerator using the following configuration:
   - Select the **Phone and Tablet** platform.
   - Choose **Blank Activity**.

2. In the `build.gradle` (Module: app) file, add the following in the dependencies section:
   ```
   compile 'com.amazonaws:aws-android-sdk-core:2.2.+'
   compile 'com.amazonaws:aws-android-sdk-lambda:2.2.+'  
   ```

3. Build the project so that the required dependencies are downloaded, as needed.

4. In the Android application manifest (`AndroidManifest.xml`), add the following permissions so that your application can connect to the Internet. You can add them just before the `<manifest>` end tag.
   ```
   <uses-permission android:name="android.permission.INTERNET" />
   <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
   ```

5. In `MainActivity`, add the following imports:
   ```
   import com.amazonaws.mobileconnectors.lambdainvoker.*;
   import com.amazonaws.auth.CognitoCachingCredentialsProvider;
   import com.amazonaws.regions.Regions;
   ```

6. In the package section, add the following two classes (RequestClass and ResponseClass). Note that the POJO is same as the POJO you created in your Lambda function in the preceding section.
   - **RequestClass**. The instances of this class act as the POJO (Plain Old Java Object) for event data which consists of first and last name. If you are using Java example for your Lambda function you created in the preceding section, this POJO is same as the POJO you created in your Lambda function code.
package com.example....lambdaeventgenerator;
public class RequestClass {
    String firstName;
    String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public RequestClass(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public RequestClass() {
    }
}

• ResponseClass

package com.example....lambdaeventgenerator;
public class ResponseClass {
    String greetings;

    public String getGreetings() {
        return greetings;
    }

    public void setGreetings(String greetings) {
        this.greetings = greetings;
    }
}
7. In the same package, create interface called `MyInterface` for invoking the `AndroidBackendLambdaFunction` Lambda function.

```java
package com.example.lambdaeventgenerator;
import com.amazonaws.mobileconnectors.lambdainvoker.LambdaFunction;
public interface MyInterface {
    /**
     * Invoke the Lambda function "AndroidBackendLambdaFunction".
     * The function name is the method name.
     */
    @LambdaFunction
    ResponseClass AndroidBackendLambdaFunction(RequestClass request);
}
```

The `@LambdaFunction` annotation in the code maps the specific client method to the same-name Lambda function.

8. To keep the application simple, we are going to add code to invoke the Lambda function in the `onCreate()` event handler. In `MainActivity`, add the following code toward the end of the `onCreate()` code.

```java
// Create an instance of CognitoCachingCredentialsProvider
CognitoCachingCredentialsProvider cognitoProvider = new
    CognitoCachingCredentialsProvider(
        this.getApplicationContext(), "identity-pool-id", Regions.US_WEST_2);

// Create LambdaInvokerFactory, to be used to instantiate the Lambda proxy.
LambdaInvokerFactory factory = new
    LambdaInvokerFactory(this.getApplicationContext(),
        Regions.US_WEST_2, cognitoProvider);
```
/ Create the Lambda proxy object with a default Json data binder.
// You can provide your own data binder by implementing
// LambdaDataBinder.
final MyInterface myInterface = factory.build(MyInterface.class);

RequestClass request = new RequestClass('John', 'Doe');
// The Lambda function invocation results in a network call.
// Make sure it is not called from the main thread.
new AsyncTask<RequestClass, Void, ResponseClass>() {
    @Override
    protected ResponseClass doInBackground(RequestClass... params) {
        // invoke "echo" method. In case it fails, it will throw a
        // LambdaFunctionException.
        try {
            return myInterface.AndroidBackendLambdaFunction(params[0]);
        } catch (LambdaFunctionException lfe) {
            Log.e("Tag", "Failed to invoke echo", lfe);
            return null;
        }
    }

    @Override
    protected void onPostExecute(ResponseClass result) {
        if (result == null) {
            return;
        }

        // Do a toast
        Toast.makeText(MainActivity.this, result.getGreetings(),
                        Toast.LENGTH_LONG).show();
    }
}.execute(request);

9. Run the code and verify it as follows:

- The Toast.makeText() displays the response returned.
- Verify that CloudWatch Logs shows the log created by the Lambda function. It should show
  the event data (first name and last name). You can also verify this in the AWS Lambda
  console.
Sample function code

Sample code is available for the following languages.

Topics
- **Node.js**
- **Java**

Node.js

The following example uses data to generate a string response.

Example index.js

```javascript
exports.handler = function(event, context, callback) {
    console.log("Received event: ", event);
    var data = {
        "greetings": "Hello, " + event.firstName + " " + event.lastName + "."
    }
    callback(null, data);
}
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Node.js Lambda functions with .zip file archives](#).

Java

The following example uses data to generate a string response.

In the code, the `handler` (myHandler) uses the RequestClass and ResponseClass types for the input and output. The code provides implementation for these types.

Example HelloPojo.java

```java
package example;

import com.amazonaws.services.lambda.runtime.Context;

public class HelloPojo {
    // Define two classes/POJOs for use with Lambda function.
}
```
public static class RequestClass {
    String firstName;
    String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public RequestClass(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public RequestClass() {
    }
}

public static class ResponseClass {
    String greetings;

    public String getGreetings() {
        return greetings;
    }

    public void setGreetings(String greetings) {
        this.greetings = greetings;
    }

    public ResponseClass(String greetings) {
        this.greetings = greetings;
    }
}

public ResponseClass() {
}

public static ResponseClass myHandler(RequestClass request, Context context) {
    String greetingString = String.format("Hello %s, %s.", request.firstName, request.lastName);
    context.getLogger().log(greetingString);
    return new ResponseClass(greetingString);
}

Dependencies

- aws-lambda-java-core

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see Deploy Java Lambda functions with .zip or JAR file archives.
Orchestrating functions with Step Functions

AWS Step Functions is an orchestration service that lets you connect Lambda functions together into serverless workflows, called state machines. Use Step Functions to orchestrate serverless applications workflows (for example, a store checkout process), build long-running workflows for IT automation and human-approval use cases, or create high-volume short-duration workflows for streaming data processing and ingestion.

Topics

- State machine application patterns
- Managing state machines in the Lambda console
- Orchestration examples with Step Functions

State machine application patterns

In Step Functions, you orchestrate your resources using state machines, which are defined using a JSON-based, structured language called Amazon States Language.

Sections

- State machine components
- State machine application patterns
- Applying patterns to state machines
- Example branching application pattern

State machine components

State machines contain elements called states that make up your workflow. The logic of each state determines which state comes next, what data to pass along, and when to terminate the workflow. A state is referred to by its name, which can be any string, but which must be unique within the scope of the entire state machine.

To create a state machine that uses Lambda, you need the following components:

1. An AWS Identity and Access Management (IAM) role for Lambda with one or more permissions policies (such as AWSLambdaRole service permissions).
2. One or more Lambda functions (with the IAM role attached) for your specific runtime.
3. A state machine authored in Amazon States Language.

**State machine application patterns**

You can create complex orchestrations for state machines using application patterns such as:

- **Catch and retry** – Handle errors using sophisticated catch-and-retry functionality.
- **Branching** – Design your workflow to choose different branches based on Lambda function output.
- **Chaining** – Connect functions into a series of steps, with the output of one step providing the input to the next step.
- **Parallelism** – Run functions in parallel, or use dynamic parallelism to invoke a function for every member of any array.

**Applying patterns to state machines**

The following shows how you can apply these application patterns to a state machine within an Amazon States Language definition.

**Catch and Retry**

A Catch field and a Retry field add catch-and-retry logic to a state machine. Catch ("Type": "Catch") is an array of objects that define a fallback state. Retry ("Type": "Retry") is an array of objects that define a retry policy if the state encounters runtime errors.

**Branching**

A Choice state adds branching logic to a state machine. Choice ("Type": "Choice") is an array of rules that determine which state the state machine transitions to next.

**Chaining**

A "Chaining" pattern describes multiple Lambda functions connected together in a state machine. You can use chaining to create reusable workflow invocations from a Task ("Type": "Task") state of a state machine.
Parallelism

A Parallel state adds parallelism logic to a state machine. You can use a **Parallel** state ("Type": "Parallel") to create parallel branches of invocation in your state machine.

Dynamic parallelism

A Map state adds dynamic "for-each" loop logic to a state machine. You can use a **Map** state ("Type": "Map") to run a set of steps for each element of an input array in a state machine.

While the Parallel state invokes multiple branches of steps using the same input, a Map state invokes the same steps for multiple entries of the array.

In addition to application patterns, Step Functions supports various service integration patterns, including the ability to pause a workflow for human approval, or to call a legacy system or other third party.

**Example branching application pattern**

In the following example, the WhichCoat state machine defined in the Amazon States Language (ASL) definition shows a branching application pattern with a **Choice** state ("Type": "Choice"). If the condition of one of the three Choice states is met, the Lambda function is invoked as a **Task**:

1. The WearHeavyCoat state invokes the `wear_heavy_coat` Lambda function and returns a message.
2. The WearLightJacket state invokes the `wear_light_jacket` Lambda function and returns a message.
3. The None state invokes the `no_jacket` Lambda function and returns a message.

The WhichCoat state machine has the following structure:
Example Example Amazon States Language definition

The following Amazon States Language definition of the WhichCoat state machine uses a Variable context object called Weather. If one of the three conditions in StringEquals is met, the Lambda function defined in the Resource field's Amazon Resource Name (ARN) is invoked.

```json
{
    "Comment":"Coat Indicator State Machine",
    "StartAt":"WhichCoat",
    "States":{
        "WhichCoat":{
            "Type":"Choice",
            "Choices":[
                {
                    "Variable":"$.Weather",
                    "StringEquals":"FREEZING",
                    "Next":"WearHeavyCoat"
                },
                {
                    "Variable":"$.Weather",
                    "StringEquals":"COOL",
                    "Next":"WearLightJacket"
                },
                {
                    "Variable":"$.Weather",
                    "StringEquals":"None",
                    "Next":"None"
                }
            ]
        }
    }
}
```
Example Example Python function

The following Lambda function in Python (wear_heavy_coat) can be invoked for the state machine defined in the previous example. If the WhichCoat state machine equals a string value of FREEZING, the wear_heavy_coat function is invoked from Lambda, and the user receives the message that corresponds with the function: "You should wear a heavy coat today."

```python
from __future__ import print_function
import datetime

def wear_heavy_coat(message, context):
    print(message)
    response = {}
```
```python
response['Weather'] = message['Weather']
response['Timestamp'] = datetime.datetime.now().strftime("%Y-%m-%d %H-%M-%S")
response['Message'] = 'You should wear a heavy coat today.'

return response
```

**Example Example invocation data**

The following input data runs the **WearHeavyCoat** state that invokes the `wear_heavy_coat` Lambda function, when the `Weather` variable is equal to a string value of `FREEZING`.

```json
{
    "Weather":"FREEZING"
}
```

For more information, see [Creating a Step Functions State Machine That Uses Lambda](https://docs.aws.amazon.com/step-functions/latest/dg/lambda-creating-sfn.html) in the *AWS Step Functions Developer Guide*.

**Managing state machines in the Lambda console**

You can use the Lambda console to edit and view details about your Step Functions state machines and the Lambda functions that they use.

**Sections**

- [Viewing state machine details](#)
- [Editing a state machine](#)
- [Running a state machine](#)

**Viewing state machine details**

The Lambda console displays a list of your state machines in the current AWS Region that contain at least one workflow step that invokes a Lambda function.

Choose a state machine to view a graphical representation of the workflow. Steps highlighted in blue represent Lambda functions. Use the graph controls to zoom in, zoom out, and center the graph.
Note

When a Lambda function is dynamically referenced with JsonPath in the state machine definition, the function details cannot be shown in the Lambda console. Instead, the function name is listed as a Dynamic reference, and the corresponding steps in the graph are grayed out.

To view state machine details

1. Open the Lambda console Step Functions state machines page.
2. Choose a state machine.
   <result>
   The Lambda console opens the Details page.
   </result>

For more information, see Step Functions in the AWS Step Functions Developer Guide.

Editing a state machine

When you want to edit a state machine, Lambda opens the Edit definition page of the Step Functions console.

To edit a state machine

1. Open the Lambda console Step Functions state machine page.
2. Choose a state machine.
3. Choose Edit.
   The Step Functions console opens the Edit definition page.
4. Edit the state machine and choose Save.

For more information about editing state machines, see Step Functions state machine language in the AWS Step Functions Developer Guide.
Running a state machine

When you want to run a state machine, Lambda opens the New execution page of the Step Functions console.

To run a state machine

1. Open the Lambda console Step Functions state machines page.
2. Choose a state machine.
3. Choose Execute.

The Step Functions console opens the New execution page.
4. (Optional) Edit the state machine and choose Start execution.

For more information about running state machines, see Step Functions state machine execution concepts in the AWS Step Functions Developer Guide.

Orchestration examples with Step Functions

All work in your Step Functions state machine is done by Tasks. A Task performs work by using an activity, a Lambda function, or by passing parameters to the API actions of other Supported AWS Service Integrations for Step Functions.

Sections

• Configuring a Lambda function as a task
• Configuring a state machine as an event source
• Handling function and service errors
• AWS CloudFormation and AWS SAM

Configuring a Lambda function as a task

Step Functions can invoke Lambda functions directly from a Task state in an Amazon States Language definition.

..."MyStateName":{
    "Type":"Task",
You can create a Task state that invokes your Lambda function with the input to the state machine or any JSON document.

**Example event.json – Input to random-error function**

```json
{
  "max-depth": 10,
  "current-depth": 0,
  "error-rate": 0.05
}
```

**Configuring a state machine as an event source**

You can create a Step Functions state machine that invokes a Lambda function. The following example shows a Task state that invokes version 1 of a function named my-function with an event payload that has three keys. When the function returns a successful response, the state machine continues to the next task.

**Example Example state machine**

```json
...
"Invoke": {
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:invoke",
  "Parameters": {
    "Payload": {
      "max-depth": 10,
      "current-depth": 0,
      "error-rate": 0.05
    }
  },
  "Next": "NEXT_STATE",
  "TimeoutSeconds": 25
}
```
Permissions

Your state machine needs permission to call the Lambda API to invoke a function. To grant it permission, add the AWS managed policy AWSLambdaRole or a function-scoped inline policy to its role. For more information, see How AWS Step Functions Works with IAM in the AWS Step Functions Developer Guide.

The FunctionName and Payload parameters map to parameters in the Invoke API operation. In addition to these, you can also specify the InvocationType and ClientContext parameters. For example, to invoke the function asynchronously and continue to the next state without waiting for a result, you can set InvocationType to Event:

"InvocationType": "Event"

Instead of hard-coding the event payload in the state machine definition, you can use the input from the state machine execution. The following example uses the input specified when you run the state machine as the event payload:

"Payload.$": "$

You can also invoke a function asynchronously and wait for it to make a callback with the AWS SDK. To do this, set the state's resource to arn:aws:states:::lambda:invoke.waitForTaskToken.

For more information, see Invoke Lambda with Step Functions in the AWS Step Functions Developer Guide.

Handling function and service errors

When your function or the Lambda service returns an error, you can retry the invocation or continue to a different state based on the error type.

The following example shows an invoke task that retries on 5XX series Lambda API exceptions (ServiceException), throttles (TooManyRequestsException), runtime errors (Lambda.Unknown), and a function-defined error named function.MaxDepthError. It also catches an error named function.DoublesRolledError and continues to a state named CaughtException when it occurs.
Example Example catch and retry pattern

```json
...
"Invoke": {
    "Type": "Task",
    "Resource": "arn:aws:states:::lambda:invoke",
    "Retry": [
        {
            "ErrorEquals": [
                "function.MaxDepthError",
                "Lambda.TooManyRequestsException",
                "Lambda.ServiceException",
                "Lambda.Unknown"
            ],
            "MaxAttempts": 5
        }
    ],
    "Catch": [
        {
            "ErrorEquals": ["function.DoublesRolledError"],
            "Next": "CaughtException"
        }
    ],
    "Parameters": {
        ...
    }
}
```

To catch or retry function errors, create a custom error type. The name of the error type must match the `errorType` in the formatted error response that Lambda returns when you throw an error.

For more information on error handling in Step Functions, see [Handling Error Conditions Using a Step Functions State Machine](https://docs.aws.amazon.com/stepfunctions/latest/dg/error-handling-examples.html) in the [AWS Step Functions Developer Guide](https://docs.aws.amazon.com/stepfunctions/latest/dg/).

**AWS CloudFormation and AWS SAM**

You can define state machines using a AWS CloudFormation template with AWS Serverless Application Model (AWS SAM). Using AWS SAM, you can define the state machine inline in the template or in a separate file. The following example shows a state machine that invokes a Lambda function that handles errors. It refers to a function resource defined in the same template (not shown).
Example Example branching pattern in template.yml

AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: An AWS Lambda application that uses AWS Step Functions.
Resources:
  statemachine:
    Type: AWS::Serverless::StateMachine
    Properties:
      DefinitionSubstitutions:
        FunctionArn: !GetAtt function.Arn
      Payload: |
        {  
          "max-depth": 5,
          "current-depth": 0,
          "error-rate": 0.2
        }
      Definition:
        StartAt: Invoke
        States:
          Invoke:
            Type: Task
            Resource: arn:aws:states:::lambda:invoke
            Parameters:
              FunctionName: "${FunctionArn}"
              Payload: "${Payload}"
              InvocationType: Event
            Retry:
              - ErrorEquals:
                - function.MaxDepthError
                - function.MaxDepthError
                - Lambda.TooManyRequestsException
                - Lambda.ServiceException
                - Lambda.Unknown
                IntervalSeconds: 1
                MaxAttempts: 5
            Catch:
              - ErrorEquals:
                - function.DoublesRolledError
                  Next: CaughtException
              - ErrorEquals:
                - States.ALL
                  Next: UncaughtException
              Next: Success
CaughtException:
  Type: Pass
  Result: The function returned an error.
  End: true
UncaughtException:
  Type: Pass
  Result: Invocation failed.
  End: true
Success:
  Type: Pass
  Result: Invocation succeeded!
  End: true
Events:
  scheduled:
    Type: Schedule
    Properties:
      Description: Run every minute
      Schedule: rate(1 minute)
  Type: STANDARD
  Policies:
    - AWSLambdaRole
      ...

This creates a state machine with the following structure:

![State Machine Diagram]

Improving startup performance with Lambda SnapStart

Lambda SnapStart for Java can improve startup performance for latency-sensitive applications by up to 10x at no extra cost, typically with no changes to your function code. The largest contributor to startup latency (often referred to as cold start time) is the time that Lambda spends initializing the function, which includes loading the function's code, starting the runtime, and initializing the function code.

With SnapStart, Lambda initializes your function when you publish a function version. Lambda takes a Firecracker microVM snapshot of the memory and disk state of the initialized execution environment, encrypts the snapshot, and caches it for low-latency access. When you invoke the function version for the first time, and as the invocations scale up, Lambda resumes new execution environments from the cached snapshot instead of initializing them from scratch, improving startup latency.

⚠️ Important

If your applications depend on uniqueness of state, you must evaluate your function code and verify that it is resilient to snapshot operations. For more information, see Handling uniqueness with Lambda SnapStart.

Supported features and limitations

SnapStart supports Java 11 and later Java managed runtimes. Other managed runtimes (such as nodejs20.x and python3.12), OS-only runtimes, and container images are not supported.

SnapStart does not support provisioned concurrency, the arm64 architecture, Amazon Elastic File System (Amazon EFS), or ephemeral storage greater than 512 MB.

To work with SnapStart, you can use the Lambda console, the AWS Command Line Interface (AWS CLI), the Lambda API, the AWS SDK for Java, AWS CloudFormation, AWS Serverless Application Model (AWS SAM), and AWS Cloud Development Kit (AWS CDK). For more information, see Activating and managing Lambda SnapStart.
Note

You can use SnapStart only on published function versions and aliases that point to versions. You can't use SnapStart on a function's unpublished version ($LATEST).

Supported Regions

SnapStart is available in the following AWS Regions:

- US East (N. Virginia)
- US East (Ohio)
- US West (N. California)
- US West (Oregon)
- Africa (Cape Town)
- Asia Pacific (Hong Kong)
- Asia Pacific (Jakarta)
- Asia Pacific (Mumbai)
- Asia Pacific (Osaka)
- Asia Pacific (Seoul)
- Asia Pacific (Singapore)
- Asia Pacific (Sydney)
- Asia Pacific (Tokyo)
- Canada (Central)
- Europe (Frankfurt)
- Europe (Ireland)
- Europe (London)
- Europe (Milan)
- Europe (Paris)
- Europe (Stockholm)
- Middle East (Bahrain)
- South America (São Paulo)
Compatibility considerations

With SnapStart, Lambda uses a single snapshot as the initial state for multiple execution environments. If your function uses any of the following during the initialization phase, then you might need to make some changes before using SnapStart:

**Uniqueness**

If your initialization code generates unique content that is included in the snapshot, then the content might not be unique when it is reused across execution environments. To maintain uniqueness when using SnapStart, you must generate unique content after initialization. This includes unique IDs, unique secrets, and entropy that's used to generate pseudorandomness. To learn how to restore uniqueness, see [Handling uniqueness with Lambda SnapStart](#).

**Network connections**

The state of connections that your function establishes during the initialization phase isn't guaranteed when Lambda resumes your function from a snapshot. Validate the state of your network connections and re-establish them as necessary. In most cases, network connections that an AWS SDK establishes automatically resume. For other connections, review the [best practices](#).

**Temporary data**

Some functions download or initialize ephemeral data, such as temporary credentials or cached timestamps, during the initialization phase. Refresh ephemeral data in the function handler before using it, even when not using SnapStart.

**SnapStart pricing**

There's no additional cost for SnapStart. You're charged based on the number of requests for your functions, the time that it takes your code to run, and the memory configured for your function. Duration is calculated from the time that your code begins running until it returns or otherwise ends, rounded up to the nearest 1 ms.

Duration charges apply to code that runs in the function handler, initialization code that's declared outside of the handler, the time it takes for the runtime (JVM) to load, and any code that runs in a runtime hook. For more information about how Lambda calculates duration, see [Monitoring for Lambda SnapStart](#).
For functions configured with SnapStart, Lambda periodically recycles the execution environments and re-runs your initialization code. For resiliency, Lambda creates snapshots in multiple Availability Zones. Charges apply each time that Lambda re-runs your initialization code in another Availability Zone. For more information about how Lambda calculates charges, see AWS Lambda Pricing.

**Comparing Lambda SnapStart and provisioned concurrency**

Both Lambda SnapStart and provisioned concurrency can reduce cold starts and outlier latencies when a function scales up. SnapStart helps you improve startup performance by up to 10x at no extra cost. Provisioned concurrency keeps functions initialized and ready to respond in double-digit milliseconds. Configuring provisioned concurrency incurs charges to your AWS account. Use provisioned concurrency if your application has strict cold start latency requirements. You can't use both SnapStart and provisioned concurrency on the same function version.

**Note**

SnapStart works best when used with function invocations at scale. Functions that are invoked infrequently might not experience the same performance improvements.

**Additional resources**

In addition to reading the other topics in this chapter, we also recommend that you try the Starting up faster with AWS Lambda SnapStart workshop and watch the Fast cold starts for your Java functions session from AWS re:Invent 2022.

**Topics**

- Activating and managing Lambda SnapStart
- Handling uniqueness with Lambda SnapStart
- Runtime hooks for Lambda SnapStart
- Monitoring for Lambda SnapStart
- Security model for Lambda SnapStart
- Best practices for working with Lambda SnapStart
Activating and managing Lambda SnapStart

To use SnapStart, activate SnapStart on a new or existing Lambda function. Then, publish and invoke a function version.

Topics

• Activating SnapStart (console)
• Activating SnapStart (AWS CLI)
• Activating SnapStart (API)
• Lambda SnapStart and function states
• Updating a snapshot
• Using SnapStart with the AWS SDK for Java
• Using SnapStart with AWS CloudFormation, AWS SAM, and AWS CDK
• Deleting snapshots

Activating SnapStart (console)

To activate SnapStart for a function

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. Choose Configuration, and then choose General configuration.
5. On the Edit basic settings page, for SnapStart, choose Published versions.
6. Choose Save.
7. Publish a function version. Lambda initializes your code, creates a snapshot of the initialized execution environment, and then caches the snapshot for low-latency access.
8. Invoke the function version.
Activating SnapStart (AWS CLI)

To activate SnapStart for an existing function

1. Update the function configuration by running the `update-function-configuration` command with the `--snap-start` option.

   ```
   aws lambda update-function-configuration \
   --function-name my-function \
   --snap-start ApplyOn=PublishedVersions
   ```

2. Publish a function version with the `publish-version` command.

   ```
   aws lambda publish-version \
   --function-name my-function
   ```

3. Confirm that SnapStart is activated for the function version by running the `get-function-configuration` command and specifying the version number. The following example specifies version 1.

   ```
   aws lambda get-function-configuration \
   --function-name my-function:1
   ```

   If the response shows that `OptimizationStatus` is On and `State` is Active, then SnapStart is activated and a snapshot is available for the specified function version.

   ```
   "SnapStart": {
     "ApplyOn": "PublishedVersions",
     "OptimizationStatus": "On"
   },
   "State": "Active",
   ```

4. Invoke the function version by running the `invoke` command and specifying the version. The following example invokes version 1.

   ```
   aws lambda invoke \
   --cli-binary-format raw-in-base64-out \
   --function-name my-function:1 \
   --payload '{ "name": "Bob" }' \
   response.json
   ```
The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/userguide/cli-aws-supported-commands.html) in the *AWS Command Line Interface User Guide for Version 2*.

### To activate SnapStart when you create a new function

1. Create a function by running the `create-function` command with the `--snap-start` option. For `--role`, specify the Amazon Resource Name (ARN) of your execution role.

   ```bash
   aws lambda create-function \
   --function-name my-function \
   --runtime "java21" \
   --zip-file fileb://my-function.zip \
   --handler my-function.handler \
   --role arn:aws:iam::111122223333:role/lambda-ex \
   --snap-start ApplyOn=PublishedVersions
   ```

2. Create a version with the `publish-version` command.

   ```bash
   aws lambda publish-version \
   --function-name my-function
   ```

3. Confirm that SnapStart is activated for the function version by running the `get-function-configuration` command and specifying the version number. The following example specifies version 1.

   ```bash
   aws lambda get-function-configuration \
   --function-name my-function:1
   ```

If the response shows that `OptimizationStatus` is On and `State` is Active, then SnapStart is activated and a snapshot is available for the specified function version. 

```json
"SnapStart": {
   "ApplyOn": "PublishedVersions",
   "OptimizationStatus": "On"
},
"State": "Active",
```
4. Invoke the function version by running the `invoke` command and specifying the version. The following example invokes version 1.

```bash
aws lambda invoke \
 --cli-binary-format raw-in-base64-out \
 --function-name my-function:1 \
 --payload '{ "name": "Bob" }' \
 response.json
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/AmazonCLI/latest/UserGuide/cli-parameters.html) in the *AWS Command Line Interface User Guide for Version 2*.

## Activating SnapStart (API)

### To activate SnapStart

1. Do one of the following:
   - Create a new function with SnapStart activated by using the `CreateFunction` API action with the `SnapStart` parameter.
   - Activate SnapStart for an existing function by using the `UpdateFunctionConfiguration` action with the `SnapStart` parameter.

2. Publish a function version with the `PublishVersion` action. Lambda initializes your code, creates a snapshot of the initialized execution environment, and then caches the snapshot for low-latency access.

3. Confirm that SnapStart is activated for the function version by using the `GetFunctionConfiguration` action. Specify a version number to confirm that SnapStart is activated for that version. If the response shows that `OptimizationStatus` is On and `State` is Active, then SnapStart is activated and a snapshot is available for the specified function version.

```json
"SnapStart": {
    "ApplyOn": "PublishedVersions",
    "OptimizationStatus": "On"
},
"State": "Active",
```
4. Invoke the function version with the **Invoke** action.

**Lambda SnapStart and function states**

The following function states can occur when you use SnapStart. They can also occur when Lambda periodically recycles the execution environment and re-runs the initialization code for a function that's configured with SnapStart.

- **Pending** – Lambda is initializing your code and taking a snapshot of the initialized execution environment. Any invocations or other API actions that operate on the function version will fail.

- **Active** – Snapshot creation is complete and you can invoke the function. To use SnapStart, you must invoke the published function version, not the unpublished version ($LATEST).

- **Inactive** – The function version hasn't been invoked for 14 days. When the function version becomes **Inactive**, Lambda deletes the snapshot. If you invoke the function version after 14 days, Lambda returns a **SnapStartNotReadyException** response and begins initializing a new snapshot. Wait until the function version reaches the **Active** state, and then invoke it again.

- **Failed** – Lambda encountered an error when running the initialization code or creating the snapshot.

**Updating a snapshot**

Lambda creates a snapshot for each published function version. To update a snapshot, publish a new function version. Lambda automatically updates your snapshots with the latest runtime and security patches.

**Using SnapStart with the AWS SDK for Java**

To make AWS SDK calls from your function, Lambda generates an ephemeral set of credentials by assuming your function's execution role. These credentials are available as environment variables during your function's invocation. You don't need to provide credentials for the SDK directly in code. By default, the credential provider chain sequentially checks each place where you can set credentials and chooses the first available—usually the environment variables (AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, and AWS_SESSION_TOKEN).
Note
When SnapStart is activated, the Java runtime automatically uses the container credentials (AWS_CONTAINER_CREDENTIALS_FULL_URI and AWS_CONTAINER_AUTHORIZATION_TOKEN) instead of the access key environment variables. This prevents credentials from expiring before the function is restored.

Using SnapStart with AWS CloudFormation, AWS SAM, and AWS CDK

- **AWS CloudFormation**: Declare the `SnapStart` entity in your template.
- **AWS Serverless Application Model (AWS SAM)**: Declare the `SnapStart` property in your template.
- **AWS Cloud Development Kit (AWS CDK)**: Use the `SnapStartProperty` type.

Deleting snapshots

Lambda deletes snapshots when:

- You delete the function or function version.
- You don't invoke the function version for 14 days. After 14 days without an invocation, the function version transitions to the `Inactive` state. If you invoke the function version after 14 days, Lambda returns a `SnapStartNotReadyException` response and begins initializing a new snapshot. Wait until the function version reaches the `Active` state, and then invoke it again.

Lambda removes all resources associated with deleted snapshots in compliance with the General Data Protection Regulation (GDPR).
Handling uniqueness with Lambda SnapStart

When invocations scale up on a SnapStart function, Lambda uses a single initialized snapshot to resume multiple execution environments. If your initialization code generates unique content that is included in the snapshot, then the content might not be unique when it is reused across execution environments. To maintain uniqueness when using SnapStart, you must generate unique content after initialization. This includes unique IDs, unique secrets, and entropy that's used to generate pseudorandomness.

We recommend the following best practices to help you maintain uniqueness in your code. Lambda also provides an open-source SnapStart scanning tool to help check for code that assumes uniqueness. If you generate unique data during the initialization phase, then you can use a runtime hook to restore uniqueness. With runtime hooks, you can run specific code immediately before Lambda takes a snapshot or immediately after Lambda resumes a function from a snapshot.

Avoid saving state that depends on uniqueness during initialization

During the initialization phase of your function, avoid caching data that's intended to be unique, such as generating a unique ID for logging. Instead, we recommend that you generate unique data inside your function handler or use a runtime hook.

Example – Generating a unique ID in function handler

The following example demonstrates how to generate a UUID in the function handler.

```java
import java.util.UUID;

public class Handler implements RequestHandler<String, String> {
    private static UUID uniqueSandboxId = null;
    @Override
    public String handleRequest(String event, Context context) {
        if (uniqueSandboxId == null)
            uniqueSandboxId = UUID.randomUUID();
        System.out.println("Unique Sandbox Id: " + uniqueSandboxId);
        return "Hello, World!";
    }
}
```
Use cryptographically secure pseudorandom number generators (CSPRNGs)

If your application depends on randomness, we recommend that you use cryptographically secure random number generators (CSPRNGs). The Lambda managed runtime for Java includes two built-in CSPRNGs (OpenSSL 1.0.2 and java.security.SecureRandom) that automatically maintain randomness with SnapStart. Software that always gets random numbers from /dev/random or /dev/urandom also maintains randomness with SnapStart.

Example – java.security.SecureRandom

The following example uses java.security.SecureRandom, which generates unique number sequences even when the function is restored from a snapshot.

```java
import java.security.SecureRandom;
public class Handler implements RequestHandler<String, String> {
    private static SecureRandom rng = new SecureRandom();
    @Override
    public String handleRequest(String event, Context context) {
        for (int i = 0; i < 10; i++) {
            System.out.println(rng.next());
        }
        return "Hello, World!";
    }
}
```

SnapStart scanning tool

Lambda provides a scanning tool to help you check for code that assumes uniqueness. The SnapStart scanning tool is an open-source SpotBugs plugin that runs a static analysis against a set of rules. The scanning tool helps identify potential code implementations that might break assumptions regarding uniqueness. For installation instructions and a list of checks that the scanning tool performs, see the aws-lambda-snapstart-java-rules repository on GitHub.

To learn more about handling uniqueness with SnapStart, see Starting up faster with AWS Lambda SnapStart on the AWS Compute Blog.
Runtime hooks for Lambda SnapStart

You can use runtime hooks to implement code before Lambda creates a snapshot or after Lambda resumes a function from a snapshot. Runtime hooks are available as part of the open-source Coordinated Restore at Checkpoint (CRaC) project. CRaC is in development for the Open Java Development Kit (OpenJDK). For an example of how to use CRaC with a reference application, see the CRaC repository on GitHub. CRaC uses three main elements:

- Resource – An interface with two methods, beforeCheckpoint() and afterRestore(). Use these methods to implement the code that you want to run before a snapshot and after a restore.
- Context <R extends Resource> – To receive notifications for checkpoints and restores, a Resource must be registered with a Context.
- Core – The coordination service, which provides the default global Context via the static method Core.getGlobalContext().

For more information about Context and Resource, see Package org.crac in the CRaC documentation.

Use the following steps to implement runtime hooks with the org.crac package. The Lambda runtime contains a customized CRaC context implementation that calls your runtime hooks before checkpointing and after restoring.

**Step 1: Update the build configuration**

Add the org.crac dependency to the build configuration. The following example uses Gradle. For examples for other build systems, see the Apache Maven documentation.

```java
dependencies {
    compile group: 'com.amazonaws', name: 'aws-lambda-java-core', version: '1.2.1'
    # All other project dependencies go here:
    # ...
    # Then, add the org.crac dependency:
    implementation group: 'io.github.crac', name: 'org-crac', version: '0.1.3'
}
```
Step 2: Update the Lambda handler

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

For more information, see [AWS Lambda function handler in Java](#).

The following example handler shows how to run code before checkpointing (`beforeCheckpoint()`) and after restoring (`afterRestore()`). This handler also registers the Resource to the runtime-managed global Context.

---

### Note

When Lambda creates a snapshot, your initialization code can run for up to 15 minutes. The time limit is 130 seconds or the configured function timeout (maximum 900 seconds), whichever is higher. Your `beforeCheckpoint()` runtime hooks count towards the initialization code time limit. When Lambda restores a snapshot, the runtime (JVM) must load and `afterRestore()` runtime hooks must complete within the timeout limit (10 seconds). Otherwise, you'll get a SnapStartTimeoutException.

---

```java
... 
import org.crac.Resource;  
import org.crac.Core;  
...
public class CRaCDemo implements RequestStreamHandler, Resource {
    public CRaCDemo() {
        Core.getGlobalContext().register(this);
    }
    public String handleRequest(String name, Context context) throws IOException {
        System.out.println("Handler execution");
        return "Hello " + name;
    }
    @Override
    public void beforeCheckpoint(org.crac.Context<? extends Resource> context)
            throws Exception {
        System.out.println("Before checkpoint");
    }
    @Override
    public void afterRestore(org.crac.Context<? extends Resource> context)
```
Context maintains only a `WeakReference` to the registered object. If a `Resource` is garbage collected, runtime hooks do not run. Your code must maintain a strong reference to the `Resource` to guarantee that the runtime hook runs.

Here are two examples of patterns to avoid:

**Example – Object without a strong reference**

```java
Core.getGlobalContext().register( new MyResource() );
```

**Example – Objects of anonymous classes**

```java
Core.getGlobalContext().register( new Resource() {
    @Override
    public void afterRestore(Context<? extends Resource> context) throws Exception {
        // ...
    }

    @Override
    public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
        // ...
    }
}
```

Instead, maintain a strong reference. In the following example, the registered resource isn't garbage collected and runtime hooks run consistently.

**Example – Object with a strong reference**

```java
Resource myResource = new MyResource(); // This reference must be maintained to prevent the registered resource from being garbage collected
Core.getGlobalContext().register( myResource );
```
Monitoring for Lambda SnapStart

You can monitor your Lambda SnapStart functions using Amazon CloudWatch, AWS X-Ray, and the Lambda Telemetry API.

ℹ️ Note

The AWS_LAMBDA_LOG_GROUP_NAME and AWS_LAMBDA_LOG_STREAM_NAME environment variables are not available in Lambda SnapStart functions.

CloudWatch for SnapStart

There are a few differences with the CloudWatch log stream format for SnapStart functions:

- **Initialization logs** – When a new execution environment is created, the REPORT doesn't include the Init Duration field. That's because Lambda initializes SnapStart functions when you create a version instead of during function invocation. For SnapStart functions, the Init Duration field is in the INIT_REPORT record. This record shows duration details for the Init phase, including the duration of any beforeCheckpoint runtime hooks.

- **Invocation logs** – When a new execution environment is created, the REPORT includes the Restore Duration and Billed Restore Duration fields:
  - Restore Duration: The time it takes for Lambda to restore a snapshot, load the runtime (JVM), and run any afterRestore runtime hooks. The process of restoring snapshots can include time spent on activities outside the MicroVM. This time is reported in Restore Duration.
  - Billed Restore Duration: The time it takes for Lambda to load the runtime (JVM) and run any afterRestore hooks. You are not charged for the time it takes to restore a snapshot.

ℹ️ Note

Duration charges apply to code that runs in the function handler, initialization code that's declared outside of the handler, the time it takes for the runtime (JVM) to load, and any code that runs in a runtime hook. For more information, see SnapStart pricing.

The cold start duration is the sum of Restore Duration + Duration.
The following example is a Lambda Insights query that returns the latency percentiles for SnapStart functions. For more information about Lambda Insights queries, see Example workflow using queries to troubleshoot a function.

```plaintext
filter @type = "REPORT"
  | parse @log /\d+:\/aws/\lambda/(?<function>\.*\)/
  | parse @message /Restore\sDuration:\s(?<restoreDuration>\.*\s)\sms/
  | stats
  count(*) as invocations,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 50) as p50,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 90) as p90,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99) as p99,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99.9) as p99.9
  group by function, (ispresent(@initDuration) or ispresent(restoreDuration)) as coldstart
  | sort by coldstart desc
```

**X-Ray active tracing for SnapStart**

You can use X-Ray to trace requests to Lambda SnapStart functions. There are a few differences with the X-Ray subsegments for SnapStart functions:

- There is no Initialization subsegment for SnapStart functions.
- The Restore subsegment shows the time it takes for Lambda to restore a snapshot, load the runtime (JVM), and run any afterRestore runtime hooks. The process of restoring snapshots can include time spent on activities outside the MicroVM. This time is reported in the Restore subsegment. You aren't charged for the time spent outside the microVM to restore a snapshot.

**Telemetry API events for SnapStart**

Lambda sends the following SnapStart events to the Telemetry API:

- `platform.restoreStart` – Shows the time when the Restore phase started.
- `platform.restoreRuntimeDone` – Shows whether the Restore phase was successful. Lambda sends this message when the runtime sends a restore/next runtime API request. There are three possible statuses: success, failure, and timeout.
- `platform.restoreReport` – Shows how long the Restore phase lasted and how many milliseconds you were billed for during this phase.
Amazon API Gateway and function URL metrics

If you create a web API using API Gateway, then you can use the IntegrationLatency metric to measure end-to-end latency (the time between when API Gateway relays a request to the backend and when it receives a response from the backend).

If you're using a Lambda function URL, then you can use the UrlRequestLatency metric to measure end-to-end latency (the time between when the function URL receives a request and when the function URL returns a response).
Security model for Lambda SnapStart

Lambda SnapStart supports encryption at rest. Lambda encrypts snapshots with an AWS KMS key. By default, Lambda uses an AWS managed key. If this default behavior suits your workflow, then you don't need to set up anything else. Otherwise, you can use the --kms-key-arn option in the create-function or update-function-configuration command to provide an AWS KMS customer managed key. You might do this to control rotation of the KMS key or to meet the requirements of your organization for managing KMS keys. Customer managed keys incur standard AWS KMS charges. For more information, see AWS Key Management Service pricing.

When you delete a SnapStart function or function version, all Invoke requests to that function or function version fail. Lambda automatically deletes snapshots that are not invoked for 14 days. Lambda removes all resources associated with deleted snapshots in compliance with the General Data Protection Regulation (GDPR).
Best practices for working with Lambda SnapStart

Topics

• Network connections
• Performance tuning

Network connections

The state of connections that your function establishes during the initialization phase isn't guaranteed when Lambda resumes your function from a snapshot. In most cases, network connections that an AWS SDK establishes automatically resume. For other connections, we recommend the following best practices.

Re-establish network connections

Always re-establish your network connections when your function resumes from a snapshot. We recommend that you re-establish network connections in the function handler. Alternatively, you can use an afterRestore runtime hook.

Don't use hostname as a unique execution environment identifier

We recommend against using hostname to identify your execution environment as a unique node or container in your applications. With SnapStart, a single snapshot is used as the initial state for multiple execution environments, and all execution environments return the same hostname value for InetAddress.getLocalHost(). For applications that require a unique execution environment identity or hostname value, we recommend that you generate a unique ID in the function handler. Or, use an afterRestore runtime hook to generate a unique ID, and then use the unique ID as the identifier for the execution environment.

Avoid binding connections to fixed source ports

We recommend that you avoid binding network connections to fixed source ports. Connections are re-established when a function resumes from a snapshot, and network connections that are bound to a fixed source port might fail.

Avoid using Java DNS cache

Lambda functions already cache DNS responses. If you use another DNS cache with SnapStart, then you might experience connection timeouts when the function resumes from a snapshot.
Performance tuning

Note

SnapStart works best when used with function invocations at scale. Functions that are invoked infrequently might not experience the same performance improvements.

To maximize the benefits of SnapStart, we recommend that you preload classes that contribute to startup latency in your initialization code instead of in the function handler. This moves the latency associated with heavy class loading out of the invocation path, optimizing startup performance with SnapStart.

If you can't preload classes during initialization, then we recommend that you preload classes with dummy invocations. To do this, update the function handler code, as shown in the following example from the pet store function on the AWS Labs GitHub repository.

```java
private static SpringLambdaContainerHandler<AwsProxyRequest, AwsProxyResponse> handler;
static {
    try {
        handler = SpringLambdaContainerHandler.getAwsProxyHandler(PetStoreSpringAppConfig.class);

        // Use the onStartup method of the handler to register the custom filter
        handler.onStartup(servletContext -> {
            FilterRegistration.Dynamic registration = servletContext.addFilter("CognitoIdentityFilter", CognitoIdentityFilter.class);
            registration.addMappingForUrlPatterns(EnumSet.of(DispatcherType.REQUEST), false, "/**");
            }

        // Send a fake Amazon API Gateway request to the handler to load classes ahead of time
        ApiGatewayRequestIdentity identity = new ApiGatewayRequestIdentity();
        identity.setApiKeyId("foo");
        identity.setAccountId("foo");
        identity.setAccessKeyId("foo");

        AwsProxyRequestContext reqCtx = new AwsProxyRequestContext();
        reqCtx.setPath("/pets");
        reqCtx.setStage("default");
```
reqCtx.setAuthorizer(null);
reqCtx.setIdentity(identity);

AwsProxyRequest req = new AwsProxyRequest();
req.setHttpMethod("GET");
req.setPath("/pets");
req.setBody("");
req.setRequestContext(reqCtx);

Context ctx = new TestContext();
handler.proxy(req, ctx);

} catch (ContainerInitializationException e) {
    // if we fail here. We re-throw the exception to force another cold start
    e.printStackTrace();
    throw new RuntimeException("Could not initialize Spring framework", e);
}
}
Lambda sample applications

The GitHub repository for this guide includes sample applications that demonstrate the use of various languages and AWS services. Each sample application includes scripts for easy deployment and cleanup, an AWS SAM template, and supporting resources.

Node.js

Sample Lambda applications in Node.js

- **blank-nodejs** – A Node.js function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.
- **nodejs-apig** – A function with a public API endpoint that processes an event from API Gateway and returns an HTTP response.
- **rds-mysql** – A function that relays queries to a MySQL for RDS Database. This sample includes a private VPC and database instance configured with a password in AWS Secrets Manager.
- **efs-nodejs** – A function that uses an Amazon EFS file system in a Amazon VPC. This sample includes a VPC, file system, mount targets, and access point configured for use with Lambda.
- **list-manager** – A function processes events from an Amazon Kinesis data stream and update aggregate lists in Amazon DynamoDB. The function stores a record of each event in a MySQL for RDS Database in a private VPC. This sample includes a private VPC with a VPC endpoint for DynamoDB and a database instance.
- **error-processor** – A Node.js function generates errors for a specified percentage of requests. A CloudWatch Logs subscription invokes a second function when an error is recorded. The processor function uses the AWS SDK to gather details about the request and stores them in an Amazon S3 bucket.

Python

Sample Lambda applications in Python

- **blank-python** – A Python function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.
Ruby

Sample Lambda applications in Ruby

- **blank-ruby** – A Ruby function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.
- **Ruby Code Samples for AWS Lambda** – Code samples written in Ruby that demonstrate how to interact with AWS Lambda.

Java

Sample Lambda applications in Java

- **java17-examples** – A Java function that demonstrates how to use a Java record to represent an input event data object.
- **java-basic** – A collection of minimal Java functions with unit tests and variable logging configuration.
- **java-events** – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the [aws-lambda-java-events](https://aws.amazon.com/sdk-for-java/) library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- **s3-java** – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- **Use API Gateway to invoke a Lambda function** – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.

Running popular Java frameworks on Lambda

- **spring-cloud-function-samples** – An example from Spring that shows how to use the [Spring Cloud Function](https://spring.io/projects/spring-cloud-function) framework to create AWS Lambda functions.
- **Serverless Spring Boot Application Demo** – An example that shows how to set up a typical Spring Boot application in a managed Java runtime with and without SnapStart, or as a GraalVM native image with a custom runtime.
• **Serverless Micronaut Application Demo** – An example that shows how to use Micronaut in a managed Java runtime with and without SnapStart, or as a GraalVM native image with a custom runtime. Learn more in the [Micronaut/Lambda guides](#).

• **Serverless Quarkus Application Demo** – An example that shows how to use Quarkus in a managed Java runtime with and without SnapStart, or as a GraalVM native image with a custom runtime. Learn more in the [Quarkus/Lambda guide](#) and [Quarkus/SnapStart guide](#).

Go

Lambda provides the following sample applications for the Go runtime:

**Sample Lambda applications in Go**

• **go-al2** – A hello world function that returns the public IP address. This app uses the provided .al2 custom runtime.

• **blank-go** – A Go function that shows the use of Lambda's Go libraries, logging, environment variables, and the AWS SDK. This app uses the go1.x runtime.

C#

**Sample Lambda applications in C#**

• **blank-csharp** – A C# function that shows the use of Lambda's .NET libraries, logging, environment variables, AWS X-Ray tracing, unit tests, and the AWS SDK.

• **blank-csharp-with-layer** – A C# function that uses the .NET CLI to create a layer that packages the function's dependencies.

• **ec2-spot** – A function that manages spot instance requests in Amazon EC2.

PowerShell

Lambda provides the following sample applications for PowerShell:

• **blank-powershell** – A PowerShell function that shows the use of logging, environment variables, and the AWS SDK.

To deploy a sample application, follow the instructions in its README file. To learn more about the architecture and use cases of an application, read the topics in this chapter.
Topics

- Blank function sample application for AWS Lambda
- Error processor sample application for AWS Lambda
- List manager sample application for AWS Lambda
Blank function sample application for AWS Lambda

The blank function sample application is a starter application that demonstrates common operations in Lambda with a function that calls the Lambda API. It shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK. Explore this application to learn about building Lambda functions in your programming language, or use it as a starting point for your own projects.

Variants of this sample application are available for the following languages:

**Variants**

- Node.js – [blank-nodejs](#).
- Python – [blank-python](#).
- Ruby – [blank-ruby](#).
- Java – [blank-java](#).
- Go – [blank-go](#).
- C# – [blank-csharp](#).
- PowerShell – [blank-powershell](#).

The examples in this topic highlight code from the Node.js version, but the details are generally applicable to all variants.

You can deploy the sample in a few minutes with the AWS CLI and AWS CloudFormation. Follow the instructions in the README to download, configure, and deploy it in your account.

**Sections**

- [Architecture and handler code](#)
- [Deployment automation with AWS CloudFormation and the AWS CLI](#)
- [Instrumentation with the AWS X-Ray](#)
- [Dependency management with layers](#)

**Architecture and handler code**

The sample application consists of function code, an AWS CloudFormation template, and supporting resources. When you deploy the sample, you use the following AWS services:
• AWS Lambda – Runs function code, sends logs to CloudWatch Logs, and sends trace data to X-Ray. The function also calls the Lambda API to get details about the account's quotas and usage in the current Region.

• AWS X-Ray – Collects trace data, indexes traces for search, and generates a service map.

• Amazon CloudWatch – Stores logs and metrics.

• AWS Identity and Access Management (IAM) – Grants permission.

• Amazon Simple Storage Service (Amazon S3) – Stores the function’s deployment package during deployment.

• AWS CloudFormation – Creates application resources and deploys function code.

Standard charges apply for each service. For more information, see AWS Pricing.

The function code shows a basic workflow for processing an event. The handler takes an Amazon Simple Queue Service (Amazon SQS) event as input and iterates through the records that it contains, logging the contents of each message. It logs the contents of the event, the context object, and environment variables. Then it makes a call with the AWS SDK and passes the response back to the Lambda runtime.

Example blank-nodejs/function/index.js – Handler code

```javascript
// Handler
exports.handler = async function(event, context) {
    event.Records.forEach(record => {
        console.log(record.body)
    })
    console.log('## ENVIRONMENT VARIABLES: ' + serialize(process.env))
    console.log('## CONTEXT: ' + serialize(context))
    console.log('## EVENT: ' + serialize(event))

    return getAccountSettings()
}

// Use SDK client
var getAccountSettings = function() {
    return lambda.getAccountSettings().promise()
}

var serialize = function(object) {
    return JSON.stringify(object, null, 2)
}
```
The input/output types for the handler and support for asynchronous programming vary per runtime. In this example, the handler method is async, so in Node.js this means that it must return a promise back to the runtime. The Lambda runtime waits for the promise to be resolved and returns the response to the invoker. If the function code or AWS SDK client return an error, the runtime formats the error into a JSON document and returns that.

The sample application doesn't include an Amazon SQS queue to send events, but uses an event from Amazon SQS (event.json) to illustrate how events are processed. To add an Amazon SQS queue to your application, see Using Lambda with Amazon SQS.

#### Deployment automation with AWS CloudFormation and the AWS CLI

The sample application's resources are defined in an AWS CloudFormation template and deployed with the AWS CLI. The project includes simple shell scripts that automate the process of setting up, deploying, invoking, and tearing down the application.

The application template uses an AWS Serverless Application Model (AWS SAM) resource type to define the model. AWS SAM simplifies template authoring for serverless applications by automating the definition of execution roles, APIs, and other resources.

The template defines the resources in the application stack. This includes the function, its execution role, and a Lambda layer that provides the function's library dependencies. The stack does not include the bucket that the AWS CLI uses during deployment or the CloudWatch Logs log group.

Example [blank-nodejs/template.yml](#) – Serverless resources

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: An AWS Lambda application that calls the Lambda API.
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs16.x
      CodeUri: function/.
      Description: Call the AWS Lambda API
```
Timeout: 10
# Function's execution role
Policies:
- AWSLambdaBasicExecutionRole
- AWSS3ReadOnlyAccess
- AWSSXrayWriteOnlyAccess
Tracing: Active
Layers:
- !Ref libs

libs:
  Type: AWS::Serverless::LayerVersion
  Properties:
    LayerName: blank-nodejs-lib
    Description: Dependencies for the blank sample app.
    ContentUri: lib/
    CompatibleRuntimes:
      - nodejs16.x

When you deploy the application, AWS CloudFormation applies the AWS SAM transform to the template to generate an AWS CloudFormation template with standard types such as AWS::Lambda::Function and AWS::IAM::Role.

Example processed template

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "An AWS Lambda application that calls the Lambda API.",
  "Resources": {
    "function": {
      "Type": "AWS::Lambda::Function",
      "Properties": {
        "Layers": [
          {
            "Ref": "libs32xmpl61b2"
          }
        ],
        "TracingConfig": {
          "Mode": "Active"
        }
      },
      "Code": {
        "S3Bucket": "lambda-artifacts-6b000xmpl1e9bf2a",
        "S3Key": "3d3axmpl1473d249d039d2d7a37512db3"
      }
    }
  }
```

Deployment automation with AWS CloudFormation and the AWS CLI
In this example, the Code property specifies an object in an Amazon S3 bucket. This corresponds to
the local path in the CodeUri property in the project template:

```
CodeUri: function/.
```

To upload the project files to Amazon S3, the deployment script uses commands in the AWS CLI.
The `cloudformation package` command preprocesses the template, uploads artifacts, and
replaces local paths with Amazon S3 object locations. The `cloudformation deploy` command
deploys the processed template with a AWS CloudFormation change set.

**Example blank-nodejs/3-deploy.sh – Package and deploy**

```bash
#!/bin/bash
set -eo pipefail
ARTIFACT_BUCKET=$(cat bucket-name.txt)
aws cloudformation package --template-file template.yml --s3-bucket $ARTIFACT_BUCKET --
output-template-file out.yml
aws cloudformation deploy --template-file out.yml --stack-name blank-nodejs --
capabilities CAPABILITY_NAMED_IAM
```

The first time you run this script, it creates a AWS CloudFormation stack named blank-nodejs. If
you make changes to the function code or template, you can run it again to update the stack.

The cleanup script (blank-nodejs/5-cleanup.sh) deletes the stack and optionally deletes the
deployment bucket and function logs.

**Instrumentation with the AWS X-Ray**

The sample function is configured for tracing with AWS X-Ray. With the tracing mode set to
active, Lambda records timing information for a subset of invocations and sends it to X-Ray. X-Ray
processes the data to generate a service map that shows a client node and two service nodes.
The first service node (AWS::Lambda) represents the Lambda service, which validates the invocation request and sends it to the function. The second node, AWS::Lambda::Function, represents the function itself.

To record additional detail, the sample function uses the X-Ray SDK. With minimal changes to the function code, the X-Ray SDK records details about calls made with the AWS SDK to AWS services.

Example **blank-nodejs/function/index.js** – Instrumentation

```javascript
const AWSXRay = require('aws-xray-sdk-core')
const AWS = AWSXRay.captureAWS(require('aws-sdk'))

// Create client outside of handler to reuse
const lambda = new AWS.Lambda()
```

Instrumenting the AWS SDK client adds an additional node to the service map and more detail in traces. In this example, the service map shows the sample function calling the Lambda API to get details about storage and concurrency usage in the current Region.

The trace shows timing details for the invocation, with subsegments for function initialization, invocation, and overhead. The invocation subsegment has a subsegment for the AWS SDK call to the GetAccountSettings API operation.

You can include the X-Ray SDK and other libraries in your function's deployment package, or deploy them separately in a Lambda layer. For Node.js, Ruby, and Python, the Lambda runtime includes the AWS SDK in the execution environment.

**Dependency management with layers**

You can install libraries locally and include them in the deployment package that you upload to Lambda, but this has its drawbacks. Larger file sizes cause increased deployment times and can prevent you from testing changes to your function code in the Lambda console. To keep the deployment package small and avoid uploading dependencies that haven't changed, the sample app creates a Lambda layer and associates it with the function.

Example **blank-nodejs/template.yml** – Dependency layer

```yaml
Resources:
  function:
    Type: AWS::Serverless::Function
```

Dependency management with layers
Properties:
Handler: index.handler
Runtime: nodejs16.x
CodeUri: function/
Description: Call the AWS Lambda API
Timeout: 10
# Function's execution role
Policies:
- AWSLambdaBasicExecutionRole
- AWSLambda_ReadOnlyAccess
- AWSXrayWriteOnlyAccess
Tracing: Active
Layers:
- !Ref libs

libs:
Type: AWS::Serverless::LayerVersion
Properties:
LayerName: blank-nodejs-lib
Description: Dependencies for the blank sample app.
ContentUri: lib/
CompatibleRuntimes:
- nodejs16.x

The `2-build-layer.sh` script installs the function's dependencies with npm and places them in a folder with the structure required by the Lambda runtime.

Example **2-build-layer.sh** – Preparing the layer

```bash
#!/bin/bash
set -eo pipefail
mkdir -p lib/nodejs
rm -rf node_modules lib/nodejs/node_modules
npm install --production
mv node_modules lib/nodejs/
```

The first time that you deploy the sample application, the AWS CLI packages the layer separately from the function code and deploys both. For subsequent deployments, the layer archive is only uploaded if the contents of the `lib` folder have changed.
Error processor sample application for AWS Lambda

The Error Processor sample application demonstrates the use of AWS Lambda to handle events from an Amazon CloudWatch Logs subscription. CloudWatch Logs lets you invoke a Lambda function when a log entry matches a pattern. The subscription in this application monitors the log group of a function for entries that contain the word ERROR. It invokes a processor Lambda function in response. The processor function retrieves the full log stream and trace data for the request that caused the error, and stores them for later use.

Function code is available in the following files:

- Random error – random-error/index.js
- Processor – processor/index.js

You can deploy the sample in a few minutes with the AWS CLI and AWS CloudFormation. To download, configure, and deploy it in your account, follow the instructions in the README.

Sections

- Architecture and event structure
- Instrumentation with AWS X-Ray
- AWS CloudFormation template and additional resources

Architecture and event structure

The sample application uses the following AWS services.

- AWS Lambda – Runs function code, sends logs to CloudWatch Logs, and sends trace data to X-Ray.
- Amazon CloudWatch Logs – Collects logs, and invokes a function when a log entry matches a filter pattern.
- AWS X-Ray – Collects trace data, indexes traces for search, and generates a service map.
- Amazon Simple Storage Service (Amazon S3) – Stores deployment artifacts and application output.

Standard charges apply for each service.
A Lambda function in the application generates errors randomly. When CloudWatch Logs detects the word ERROR in the function's logs, it sends an event to the processor function for processing.

**Example CloudWatch Logs message event**

```json
{
    "awslogs": {
        "data": "H4sIAAAAAAAAAHWQT0/DMAzFv0vEkbLYcdJkt4qVXmCDteIAm1DbZKjS+kdpB0Jo350MhsQFyVLsZ+unl/fJWje05asrPgH5..."
    }
}
```

When it's decoded, the data contains details about the log event. The function uses these details to identify the log stream, and parses the log message to get the ID of the request that caused the error.

**Example decoded CloudWatch Logs event data**

```json
{
    "messageType": "DATA_MESSAGE",
    "owner": "123456789012",
    "logGroup": "/aws/lambda/lambda-error-processor-randomerror-1GD4SSDNACNP4",
    "logStream": "/aws/lambda/lambda-error-processor-randomerror-1GD4SSDNACNP4/2019/04/04/$LATEST/63311769a9d742f19cedf8d2e38995b9",
    "subscriptionFilters": [
        "lambda-error-processor-subscription-15OPDVQ59CG07"
    ],
    "logEvents": [
        {
            "id": "34664632210239891980253245280462376874059932423703429141",
            "timestamp": 1554415868243,
            "message": "2019-04-04T22:11:08.243Z\t1d2c1444-efd1-43ec-b16e-8fb2d37508b8\tERROR\n"
        }
    ]
}
```

The processor function uses information from the CloudWatch Logs event to download the full log stream and X-Ray trace for a request that caused an error. It stores both in an Amazon S3 bucket. To allow the log stream and trace time to finalize, the function waits for a short period of time before accessing the data.
Instrumentation with AWS X-Ray

The application uses AWS X-Ray to trace function invocations and the calls that functions make to AWS services. X-Ray uses the trace data that it receives from functions to create a service map that helps you identify errors.

The two Node.js functions are configured for active tracing in the template, and are instrumented with the AWS X-Ray SDK for Node.js in code. With active tracing, Lambda tags adds a tracing header to incoming requests and sends a trace with timing details to X-Ray. Additionally, the random error function uses the X-Ray SDK to record the request ID and user information in annotations. The annotations are attached to the trace, and you can use them to locate the trace for a specific request.

The processor function gets the request ID from the CloudWatch Logs event, and uses the AWS SDK for JavaScript to search X-Ray for that request. It uses AWS SDK clients, which are instrumented with the X-Ray SDK, to download the trace and log stream. Then it stores them in the output bucket. The X-Ray SDK records these calls, and they appear as subsegments in the trace.

AWS CloudFormation template and additional resources

The application is implemented in two Node.js modules and deployed with an AWS CloudFormation template and shell scripts. The template creates the processor function, the random error function, and the following supporting resources.

- Execution role – An IAM role that grants the functions permission to access other AWS services.
- Primer function – An additional function that invokes the random error function to create a log group.
- Custom resource – An AWS CloudFormation custom resource that invokes the primer function during deployment to ensure that the log group exists.
- CloudWatch Logs subscription – A subscription for the log stream that triggers the processor function when the word ERROR is logged.
- Resource-based policy – A permission statement on the processor function that allows CloudWatch Logs to invoke it.
- Amazon S3 bucket – A storage location for output from the processor function.

View the application template on GitHub.
To work around a limitation of Lambda's integration with AWS CloudFormation, the template creates an additional function that runs during deployments. All Lambda functions come with a CloudWatch Logs log group that stores output from function executions. However, the log group isn't created until the function is invoked for the first time.

To create the subscription, which depends on the existence of the log group, the application uses a third Lambda function to invoke the random error function. The template includes the code for the primer function inline. An AWS CloudFormation custom resource invokes it during deployment. DependsOn properties ensure that the log stream and resource-based policy are created prior to the subscription.
List manager sample application for AWS Lambda

The list manager sample application demonstrates the use of AWS Lambda to process records in an Amazon Kinesis data stream. A Lambda event source mapping reads records from the stream in batches and invokes a Lambda function. The function uses information from the records to update documents in Amazon DynamoDB and stores the records it processes in Amazon Relational Database Service (Amazon RDS).

Clients send records to a Kinesis stream, which stores them and makes them available for processing. The Kinesis stream is used like a queue to buffer records until they can be processed. Unlike an Amazon SQS queue, records in a Kinesis stream are not deleted after they are processed, so multiple consumers can process the same data. Records in Kinesis are also processed in order, where queue items can be delivered out of order. Records are deleted from the stream after 7 days.

In addition to the function that processes events, the application includes a second function for performing administrative tasks on the database. Function code is available in the following files:

- Processor – processor/index.js
- Database admin – dbadmin/index.js

You can deploy the sample in a few minutes with the AWS CLI and AWS CloudFormation. To download, configure, and deploy it in your account, follow the instructions in the README.

Sections

- Architecture and event structure
- Instrumentation with AWS X-Ray
- AWS CloudFormation templates and additional resources

Architecture and event structure

The sample application uses the following AWS services:

- Kinesis – Receives events from clients and stores them temporarily for processing.
- AWS Lambda – Reads from the Kinesis stream and sends events to the function’s handler code.
- DynamoDB – Stores lists generated by the application.
- Amazon RDS – Stores a copy of processed records in a relational database.
• AWS Secrets Manager – Stores the database password.
• Amazon VPC – Provides a private local network for communication between the function and database.

**Pricing**

Standard charges apply for each service.

The application processes JSON documents from clients that contain information necessary to update a list. It supports two types of list: tally and ranking. A *tally* contains values that are added to the current value for key if it exists. Each entry processed for a user increases the value of a key in the specified table.

The following example shows a document that increases the `xp` (experience points) value for a user's `stats` list.

**Example record – Tally type**

```json
{
    "title": "stats",
    "user": "bill",
    "type": "tally",
    "entries": {
        "xp": 83
    }
}
```

A *ranking* contains a list of entries where the value is the order in which they are ranked. A ranking can be updated with different values that overwrite the current value, instead of incrementing it. The following example shows a ranking of favorite movies:

**Example record – Ranking type**

```json
{
    "title": "favorite movies",
    "user": "mike",
    "type": "rank",
    "entries": {
```

Architecture and event structure
A Lambda event source mapping read records from the stream in batches and invokes the processor function. The event that the function handler received contains an array of objects that each contain details about a record, such as when it was received, details about the stream, and an encoded representation of the original record document.

**Example**  
**events/kinesis.json** – Record

```json
{
  "Records": [
    {
      "kinesis": {
        "kinesisSchemaVersion": "1.0",
        "partitionKey": "0",
        "sequenceNumber": "49598630142999655949899443842509554952738656579378741250",
        "data": "eyJ0aXRsZSI6ICJmYXZvcml0ZSBtb3ZpZXcifSwiInR5cGUiOiAicmFuayIsICJlbnRyaWVzIjogeyJibGFkZSBydW5uZXIiOiAyLCAidGhlIGVtcGlyZSBzdHJpa2VzIGJhY2siOiAzLCAiYWxpZW4iOiAxfX0=",
        "approximateArrivalTimestamp": 1570667770.615
      },
      "eventSource": "aws:kinesis",
      "eventVersion": "1.0",
      "eventID": "shardId-000000000000:49598630142999655949899443842509554952738656579378741250",
      "eventName": "aws:kinesis:record",
      "invokeIdentityArn": "arn:aws:iam::123456789012:role/list-manager-processorRole-7FYXMPFH7IUS",
      "awsRegion": "us-east-2",
      "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/list-manager-stream-87B3XMPF1AZ"
    },
    ...
  ]
}
```

When it's decoded, the data contains a record. The function uses the record to update the user's list and an aggregate list that stores accumulated values across all users. It also stores a copy of the event in the application's database.
Instrumentation with AWS X-Ray

The application uses AWS X-Ray to trace function invocations and the calls that functions make to AWS services. X-Ray uses the trace data that it receives from functions to create a service map that helps you identify errors.

The Node.js function is configured for active tracing in the template, and is instrumented with the AWS X-Ray SDK for Node.js in code. The X-Ray SDK records a subsegment for each call made with an AWS SDK or MySQL client.

The function uses the AWS SDK for JavaScript in Node.js to read and write to two tables for each record. The primary table stores the current state of each combination of list name and user. The aggregate table stores lists that combine data from multiple users.

AWS CloudFormation templates and additional resources

The application is implemented in Node.js modules and deployed with an AWS CloudFormation template and shell scripts. The application template creates two functions, a Kinesis stream, DynamoDB tables and the following supporting resources.

Application resources

- Execution role – An IAM role that grants the functions permission to access other AWS services.
- Lambda event source mapping – Reads records from the data stream and invokes the function.

View the application template on GitHub.

A second template, template-vpcrds.yml, creates the Amazon VPC and database resources. While it is possible to create all of the resources in one template, separating them makes it easier to clean up the application and allows the database to be reused with multiple applications.

Infrastructure resources

- VPC – A virtual private cloud network with private subnets, a route table, and a VPC endpoint that allows the function to communicate with DynamoDB without an internet connection.
- Database – An Amazon RDS database instance and a subnet group that connects it to the VPC.
# Using Lambda with an AWS SDK

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples, and documentation that make it easier for developers to build applications in their preferred language.

<table>
<thead>
<tr>
<th>SDK documentation</th>
<th>Code examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS SDK for C++</td>
<td>AWS SDK for C++ code examples</td>
</tr>
<tr>
<td>AWS SDK for Go</td>
<td>AWS SDK for Go code examples</td>
</tr>
<tr>
<td>AWS SDK for Java</td>
<td>AWS SDK for Java code examples</td>
</tr>
<tr>
<td>AWS SDK for JavaScript</td>
<td>AWS SDK for JavaScript code examples</td>
</tr>
<tr>
<td>AWS SDK for Kotlin</td>
<td>AWS SDK for Kotlin code examples</td>
</tr>
<tr>
<td>AWS SDK for .NET</td>
<td>AWS SDK for .NET code examples</td>
</tr>
<tr>
<td>AWS SDK for PHP</td>
<td>AWS SDK for PHP code examples</td>
</tr>
<tr>
<td>AWS SDK for Python (Boto3)</td>
<td>AWS SDK for Python (Boto3) code examples</td>
</tr>
<tr>
<td>AWS SDK for Ruby</td>
<td>AWS SDK for Ruby code examples</td>
</tr>
<tr>
<td>AWS SDK for Rust</td>
<td>AWS SDK for Rust code examples</td>
</tr>
<tr>
<td>AWS SDK for SAP ABAP</td>
<td>AWS SDK for SAP ABAP code examples</td>
</tr>
<tr>
<td>AWS SDK for Swift</td>
<td>AWS SDK for Swift code examples</td>
</tr>
</tbody>
</table>

For examples specific to Lambda, see [Code examples for Lambda using AWS SDKs](#).

**Example availability**

Can't find what you need? Request a code example by using the Provide feedback link at the bottom of this page.
Code examples for Lambda using AWS SDKs

The following code examples show how to use Lambda with an AWS software development kit (SDK).

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Cross-service examples are sample applications that work across multiple AWS services.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Get started

Hello Lambda

The following code examples show how to get started using Lambda.

.NET

AWS SDK for .NET

```
namespace LambdaActions;

using Amazon.Lambda;

public class HelloLambda
{

```
static async Task Main(string[] args)
{
    var lambdaClient = new AmazonLambdaClient();

    Console.WriteLine("Hello AWS Lambda");
    Console.WriteLine("Let's get started with AWS Lambda by listing your existing Lambda functions:");

    var response = await lambdaClient.ListFunctionsAsync();
    response.Functions.ForEach(function =>
    {
        Console.WriteLine($"{function.FunctionName}\t{function.Description}");
    });
}

• For API details, see ListFunctions in AWS SDK for .NET API Reference.

C++

SDK for C++

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Code for the CMakeLists.txt CMake file.

# Set the minimum required version of CMake for this project.
cmake_minimum_required(VERSION 3.13)

# Set the AWS service components used by this project.
set(SERVICE_COMPONENTS lambda)

# Set this project's name.
project("hello_lambda")
# Set the C++ standard to use to build this target.
# At least C++ 11 is required for the AWS SDK for C++.
set(CMAKE_CXX_STANDARD 11)

# Use the MSVC variable to determine if this is a Windows build.
set(WINDOWS_BUILD ${MSVC})

if (WINDOWS_BUILD)
  # Set the location where CMake can find the installed
  # libraries for the AWS SDK.
  string(REPLACE ";" "/aws-cpp-sdk-all;" SYSTEM_MODULE_PATH
"${CMAKE_SYSTEM_PREFIX_PATH}/aws-cpp-sdk-all")
  list(APPEND CMAKE_PREFIX_PATH ${SYSTEM_MODULE_PATH})
endif()

# Find the AWS SDK for C++ package.
find_package(AWSSDK REQUIRED COMPONENTS ${SERVICE_COMPONENTS})

if (WINDOWS_BUILD)
  # Copy relevant AWS SDK for C++ libraries into the current binary directory
  # for running and debugging.
  # set(BIN_SUB_DIR "/Debug") # if you are building from the command line you
  # may need to uncomment this
  # and set the proper subdirectory to the
  # executables' location.

  AWSSDK_CPY_DYN_LIBS(SERVICE_COMPONENTS ""
"${CMAKE_CURRENT_BINARY_DIR}${BIN_SUB_DIR}")
endif()

add_executable(${PROJECT_NAME}
  hello_lambda.cpp)

target_link_libraries(${PROJECT_NAME}
  ${AWSSDK_LINK_LIBRARIES})

Code for the hello_lambda.cpp source file.

#include <aws/core/Aws.h>
#include <aws/lambda/LambdaClient.h>
#include <aws/lambda/model/ListFunctionsRequest.h>
#include <iostream>
int main(int argc, char **argv) {
    Aws::SDKOptions options;
    // Optionally change the log level for debugging.
    // options.loggingOptions.logLevel = Utils::Logging::LogLevel::Debug;
    Aws::InitAPI(options); // Should only be called once.
    int result = 0;
    {
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

        Aws::Lambda::LambdaClient lambdaClient(clientConfig);
        std::vector<Aws::String> functions;
        Aws::String marker; // Used for pagination.

        do {
            Aws::Lambda::Model::ListFunctionsRequest request;
            if (!marker.empty()) {
                request.SetMarker(marker);
            }

            Aws::Lambda::Model::ListFunctionsOutcome outcome = lambdaClient.ListFunctions(
                request);

            if (outcome.IsSuccess()) {
                const Aws::Lambda::Model::ListFunctionsResult &listFunctionsResult = outcome.GetResult();
                std::cout << listFunctionsResult.GetFunctions().size() << " lambda functions were retrieved."
                         << std::endl;

                for (const Aws::Lambda::Model::FunctionConfiguration &functionConfiguration: listFunctionsResult.GetFunctions()) {

```
functions.push_back(functionConfiguration.GetFunctionName());
std::cout << functions.size() << " " << functionConfiguration.GetDescription() << std::endl;
std::cout << " " << Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(functionConfiguration.GetRuntime()) << ": " << functionConfiguration.GetHandler() << std::endl;
}
marker = listFunctionsResult.GetNextMarker();
} else {
    std::cerr << "Error with Lambda::ListFunctions. " << outcome.GetError().GetMessage() << std::endl;
    result = 1;
    break;
}
} while (!marker.empty());
}

Aws::ShutdownAPI(options); // Should only be called once.
return result;

- For API details, see [ListFunctions](https://docs.aws.amazon.com/sdk-for-cpp/v1/developer-guide/lam-operations-listfunctions.html) in [AWS SDK for C++ API Reference](https://docs.aws.amazon.com/sdk-for-cpp/v1/developer-guide/).

**Go**

**SDK for Go V2**

- **Note**
  
  There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-lambda-function-for-go).
import (   "context"   "fmt"
)

// main uses the AWS SDK for Go (v2) to create an AWS Lambda client and list up to 10
// functions in your account.
// This example uses the default settings specified in your shared credentials
// and config files.
func main() {
  sdkConfig, err := config.LoadDefaultConfig(context.TODO())
  if err != nil {
    fmt.Println("Couldn't load default configuration. Have you set up your AWS account?")
    fmt.Println(err)
    return
  }
  lambdaClient := lambda.NewFromConfig(sdkConfig)

  maxItems := 10
  fmt.Printf("Let's list up to %v functions for your account.\n", maxItems)
  result, err := lambdaClient.ListFunctions(context.TODO(),
                                          &lambda.ListFunctionsInput{
    MaxItems: aws.Int32(int32(maxItems)),
  })
  if err != nil {
    fmt.Printf("Couldn't list functions for your account. Here's why: %v\n", err)
    return
  }
  if len(result.Functions) == 0 {
    fmt.Println("You don't have any functions!"
  } else {
    for _, function := range result.Functions {
      fmt.Printf("%v\n", *function.FunctionName)
    }
  }
}
- For API details, see ListFunctions in AWS SDK for Go API Reference.

Java

SDK for Java 2.x

```java
package com.example.lambda;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.LambdaException;
import software.amazon.awssdk.services.lambda.model.ListFunctionsResponse;
import software.amazon.awssdk.services.lambda.model.FunctionConfiguration;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 */
public class ListLambdaFunctions {
    public static void main(String[] args) {
        Region region = Region.US_WEST_2;
        LambdaClient awsLambda = LambdaClient.builder()
            .region(region)
            .build();

        listFunctions(awsLambda);
        awsLambda.close();
    }
}
```

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
public static void listFunctions(LambdaClient awsLambda) {
    try {
        ListFunctionsResponse functionResult = awsLambda.listFunctions();
        List<FunctionConfiguration> list = functionResult.functions();
        for (FunctionConfiguration config : list) {
            System.out.println("The function name is " +
            config.functionName());
        }
    } catch (LambdaException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}

- For API details, see [ListFunctions](#) in *AWS SDK for Java 2.x API Reference*.

**JavaScript**

**SDK for JavaScript (v3)**

```javascript
import { LambdaClient, paginateListFunctions } from "@aws-sdk/client-lambda";

const client = new LambdaClient({});

export const helloLambda = async () => {
    const paginator = paginateListFunctions({ client }, {});
    const functions = [];
    for await (const page of paginator) {
        const funcNames = page.Functions.map((f) => f.FunctionName);
        functions.push(...funcNames);
    }
```

*Note*

There's more on GitHub. Find the complete example and learn how to set up and run in the *AWS Code Examples Repository*.  

```javascript
import { LambdaClient, paginateListFunctions } from "@aws-sdk/client-lambda";

const client = new LambdaClient({});

export const helloLambda = async () => {
    const paginator = paginateListFunctions({ client }, {});
    const functions = [];
    for await (const page of paginator) {
        const funcNames = page.Functions.map((f) => f.FunctionName);
        functions.push(...funcNames);
    }
    ```
For API details, see ListFunctions in AWS SDK for JavaScript API Reference.

Code examples

- **Actions for Lambda using AWS SDKs**
  - Create a Lambda function using an AWS SDK
  - Delete a Lambda function using an AWS SDK
  - Get a Lambda function using an AWS SDK
  - Invoke a Lambda function using an AWS SDK
  - List Lambda functions using an AWS SDK
  - Update Lambda function code using an AWS SDK
  - Update Lambda function configuration using an AWS SDK

- **Scenarios for Lambda using AWS SDKs**
  - Get started creating and invoking Lambda functions using an AWS SDK

- **Serverless examples for Lambda using AWS SDKs**
  - Invoke a Lambda function from a Kinesis trigger
  - Invoke a Lambda function from an Amazon S3 trigger
  - Invoke a Lambda function from an Amazon SNS trigger
  - Invoke a Lambda function from an Amazon SQS trigger
  - Reporting batch item failures for Lambda functions with a Kinesis trigger
  - Reporting batch item failures for Lambda functions with an Amazon SQS trigger

- **Cross-service examples for Lambda using AWS SDKs**
  - Create an API Gateway REST API to track COVID-19 data
  - Create a lending library REST API
  - Create a messenger application with Step Functions
  - Create a photo asset management application that lets users manage photos using labels
  - Create a websocket chat application with API Gateway
• Create an application that analyzes customer feedback and synthesizes audio
• Invoke a Lambda function from a browser
• Use API Gateway to invoke a Lambda function
• Use Step Functions to invoke Lambda functions
• Use scheduled events to invoke a Lambda function

Actions for Lambda using AWS SDKs

The following code examples demonstrate how to perform individual Lambda actions with AWS SDKs. These excerpts call the Lambda API and are code excerpts from larger programs that must be run in context. Each example includes a link to GitHub, where you can find instructions for setting up and running the code.

The following examples include only the most commonly used actions. For a complete list, see the AWS Lambda API Reference.

Examples
• Create a Lambda function using an AWS SDK
• Delete a Lambda function using an AWS SDK
• Get a Lambda function using an AWS SDK
• Invoke a Lambda function using an AWS SDK
• List Lambda functions using an AWS SDK
• Update Lambda function code using an AWS SDK
• Update Lambda function configuration using an AWS SDK

Create a Lambda function using an AWS SDK

The following code examples show how to create a Lambda function.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

• Get started with functions
Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```csharp
/// <summary>
/// Creates a new Lambda function.
/// </summary>
/// <param name="functionName">The name of the function.</param>
/// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3) bucket where the zip file containing the code is located.</param>
/// <param name="s3Key">The Amazon S3 key of the zip file.</param>
/// <param name="role">The Amazon Resource Name (ARN) of a role with the appropriate Lambda permissions.</param>
/// <param name="handler">The name of the handler function.</param>
/// <returns>The Amazon Resource Name (ARN) of the newly created Lambda function.</returns>
public async Task<string> CreateLambdaFunctionAsync(
    string functionName,
    string s3Bucket,
    string s3Key,
    string role,
    string handler)
{
    // Defines the location for the function code.
    // S3Bucket - The S3 bucket where the file containing the source code is stored.
    // S3Key - The name of the file containing the code.
    var functionCode = new FunctionCode
    {
        S3Bucket = s3Bucket,
        S3Key = s3Key,
    };

    var createFunctionRequest = new CreateFunctionRequest
    {
        FunctionName = functionName,
```
Description = "Created by the Lambda .NET API",
Code = functionCode,
Handler = handler,
Runtime = Runtime.Dotnet6,
Role = role,

};

var reponse = await
_lambdaService.CreateFunctionAsync(createFunctionRequest);
return reponse.FunctionArn;

• For API details, see CreateFunction in AWS SDK for .NET API Reference.

C++

SDK for C++

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Aws::Client::ClientConfiguration clientConfig;
// Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
// clientConfig.region = "us-east-1";

Aws::Lambda::LambdaClient client(clientConfig);

Aws::Lambda::Model::CreateFunctionRequest request;
request.SetFunctionName(LAMBDA_NAME);
request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
request.SetTimeout(15);
request.SetMemorySize(128);
// Assume the AWS Lambda function was built in Docker with same architecture
// as this code.
#if defined(__x86_64__)
    request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
    request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
    #error "Unimplemented architecture"
#endif // defined(architecture)
#else
    request.SetRuntime(Aws::Lambda::Model::Runtime::python3_8);
#endif
request.SetRole(roleArn);
request.SetHandler(LAMBDA_HANDLER_NAME);
request.SetPublish(true);
Aws::Lambda::Model::FunctionCode code;
std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
    std::ios_base::in | std::ios_base::binary);
if (!ifstream.is_open()) {
    std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." << std::endl;
#if USE_CPP_LAMBDA_FUNCTION
    std::cerr
        << "The cpp Lambda function must be built following the instructions in the cpp_lambda/README.md file. " << std::endl;
#endif
    deleteIamRole(clientConfig);
    return false;
}
Aws::StringStream buffer;
buffer << ifstream.rdbuf();

code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
    buffer.str().c_str(),
    buffer.str().length()));
request.SetCode(code);

Aws::Lambda::Model::CreateFunctionOutcome outcome =
    client.CreateFunction(
        request);
if (outcome.IsSuccess()) {
    std::cout << "The lambda function was successfully created. " << seconds << " seconds elapsed." << std::endl;
    break;
}
else {
    std::cerr << "Error with CreateFunction. "
             << outcome.GetError().GetMessage() << std::endl;
    deleteIamRole(clientConfig);
    return false;
}

- For API details, see CreateFunction in AWS SDK for C++ API Reference.

CLI

AWS CLI version 2

To create a Lambda function

The following create-function example creates a Lambda function named my-function.

```bash
aws lambda create-function \
    --function-name my-function \
    --runtime nodejs14.x \ 
    --zip-file fileb://my-function.zip \ 
    --handler my-function.handler \ 
    --role arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-tges6bf4
```

Contents of my-function.zip: This file is a deployment package that contains your function code and any dependencies.

Output:

```json
{
```
For more information, see [AWS Lambda Function Configuration](https://docs.aws.amazon.com/lambda/latest/dg/configuration-console.html) in the *AWS Lambda Developer Guide*.


Go

**SDK for Go V2**

```go
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
    LambdaClient *lambda.Client
}
```
// CreateFunction creates a new Lambda function from code contained in the
// zipPackage. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
// until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(functionName string, handlerName
// string, iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
var state types.State
_, err := wrapper.LambdaClient.CreateFunction(context.TODO(),
&lambda.CreateFunctionInput{
  FunctionName: aws.String(functionName),
  Role: iamRoleArn,
  Handler: aws.String(handlerName),
  Publish: true,
  Runtime: types.RuntimePython38,
})
if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
    log.Printf("Function %v already exists.\n", functionName)
    state = types.StateActive
  } else {
    log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
  }
} else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(context.TODO(),
&lambda.GetFunctionInput{
  FunctionName: aws.String(functionName)), 1*time.Minute)
if err != nil {
  log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
functionName, err)
} else {
  state = funcOutput.Configuration.State
}
}
return state
• For API details, see CreateFunction in AWS SDK for Go API Reference.

Java

SDK for Java 2.x

import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.CreateFunctionRequest;
import software.amazon.awssdk.services.lambda.model.FunctionCode;
import software.amazon.awssdk.services.lambda.model.CreateFunctionResponse;
import software.amazon.awssdk.services.lambda.model.GetFunctionRequest;
import software.amazon.awssdk.services.lambda.model.GetFunctionResponse;
import software.amazon.awssdk.services.lambda.model.LambdaException;
import software.amazon.awssdk.services.lambda.model.Runtime;
import software.amazon.awssdk.services.lambda.waiters.LambdaWaiter;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;

/**
 * This code example requires a ZIP or JAR that represents the code of the
 * Lambda function.
 * If you do not have a ZIP or JAR, please refer to the following document:
 *  
 * https://github.com/aws-doc-sdk-examples/tree/master/javav2/usecases/
 * creating_workflows_stepfunctions
 *  
 * Also, set up your development environment, including your credentials.
 */
Create a function

```java
public class CreateFunction {
    public static void main(String[] args) {
        final String usage = ""
        Usage:
            <functionName> <filePath> <role> <handler>
        Where:
            functionName - The name of the Lambda function.
            filePath - The path to the ZIP or JAR where the code is located.
            role - The role ARN that has Lambda permissions.
            handler - The fully qualified method name (for example, example.Handler::handleRequest).
        "\n
        if (args.length != 4) {
            System.out.println(usage);
            System.exit(1);
        }

        String functionName = args[0];
        String filePath = args[1];
        String role = args[2];
        String handler = args[3];
        Region region = Region.US_WEST_2;
        LambdaClient awsLambda = LambdaClient.builder()
            .region(region)
            .build();

        createLambdaFunction(awsLambda, functionName, filePath, role, handler);
        awsLambda.close();
    }

    public static void createLambdaFunction(LambdaClient awsLambda,
        String functionName,
        String filePath,
        String role,
        String handler) {
    }
}
```
try {
    LambdaWaiter waiter = awsLambda.waiter();
    InputStream is = new FileInputStream(filePath);
    SdkBytes fileToUpload = SdkBytes.fromInputStream(is);

    FunctionCode code = FunctionCode.builder()
                     .zipFile(fileToUpload)
                     .build();

    CreateFunctionRequest functionRequest =
        CreateFunctionRequest.builder()
                     .functionName(functionName)
                     .description("Created by the Lambda Java API")
                     .code(code)
                     .handler(handler)
                     .runtime(Runtime.JAVA8)
                     .role(role)
                     .build();

    // Create a Lambda function using a waiter.
    CreateFunctionResponse functionResponse =
        awsLambda.createFunction(functionRequest);
    GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
                     .functionName(functionName)
                     .build();
    WaiterResponse<GetFunctionResponse> waiterResponse =
        waiter.waitUntilFunctionExists(getFunctionRequest);
    waiterResponse.matched().response().ifPresent(System.out::println);
    System.out.println("The function ARN is " +
                        functionResponse.functionArn());
}

} catch (LambdaException | FileNotFoundException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}
JavaScript

SDK for JavaScript (v3)

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- For API details, see [CreateFunction](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/Lambda.html#createFunction-property) in AWS SDK for JavaScript API Reference.

Kotlin

SDK for Kotlin

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
suspend fun createNewFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String
): String? {

    val functionCode = FunctionCode {
        s3Bucket = s3BucketName
        s3Key = myS3Key
    }

    val request = CreateFunctionRequest {
        functionName = myFunctionName
        code = functionCode
        description = "Created by the Lambda Kotlin API"
        handler = myHandler
        role = myRole
        runtime = Runtime.Java8
    }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitForFunctionActive {
            functionName = myFunctionName
        }
        return functionResponse.functionArn
    }
}

• For API details, see [CreateFunction](#) in *AWS SDK for Kotlin API reference*.
**SDK for PHP**

```php
public function createFunction($functionName, $role, $bucketName, $handler)
{
    //This assumes the Lambda function is in an S3 bucket.
    return $this->customWaiter(function () use ($functionName, $role, $bucketName, $handler) {
        return $this->lambdaClient->createFunction(
            'Code' => [
                'S3Bucket' => $bucketName,
                'S3Key' => $functionName,
            ],
            'FunctionName' => $functionName,
            'Role' => $role['Arn'],
            'Runtime' => 'python3.9',
            'Handler' => "$handler.lambda_handler",
        );
    });
}
```

- For API details, see [CreateFunction](https://docs.aws.amazon.com/lambda/developerguide/createlambdafunction.html) in [AWS SDK for PHP API Reference](https://docs.aws.amazon.com/aws-sdk-php/v3/api/class-LambdaClient#createFunction)

**Python**

**SDK for Python (Boto3)**

```python
def create_function(function_name, role, bucket_name, handler):
    # This assumes the Lambda function is in an S3 bucket.
    return boto3.client('lambda').create_function(
        Code=dict(S3Bucket=bucket_name, S3Key=function_name),
        FunctionName=function_name,
        Role=role['Arn'],
        Runtime='python3.9',
        Handler='$handler.lambda_handler'
    )
```

class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource

    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
    ):
        """
        Deploys a Lambda function.

        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function. This
            must include the file name and the function name.
        :param iam_role: The IAM role to use for the function.
        :param deployment_package: The deployment package that contains the
            function code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
                Runtime="python3.8",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={'ZipFile': deployment_package},
                Publish=True,
            )
            function_arn = response['FunctionArn']
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response['FunctionArn'],
            )
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
else:
    return function_arn

- For API details, see CreateFunction in AWS SDK for Python (Boto3) API Reference.

Ruby

**SDK for Ruby**

```ruby
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Deploys a Lambda function.
  #
  # @param function_name: The name of the Lambda function.
  # @param handler_name: The fully qualified name of the handler function. This
  #   must include the file name and the function name.
  # @param role_arn: The IAM role to use for the function.
  # @param deployment_package: The deployment package that contains the function
  #   code in .zip format.
  # @return: The Amazon Resource Name (ARN) of the newly created function.
  def create_function(function_name, handler_name, role_arn, deployment_package)
    response = @lambda_client.create_function({
      role: role_arn.to_s,
      function_name: function_name,
      handler: handler_name,
      runtime: "ruby2.7",
      code: {
```
**Create Function**

```rust
/**
 * Create a function, uploading from a zip file.
 */
pub async fn create_function(&self, zip_file: PathBuf) -> Result<String, anyhow::Error> {
    let code = self.prepare_function(zip_file, None).await?
    let key = code.s3_key().unwrap().to_string();
```

- For API details, see [CreateFunction](#) in *AWS SDK for Ruby API Reference*.

**Rust**

**SDK for Rust**

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the *AWS Code Examples Repository*.
self.create_role().await;

let role = self
    .iam_client
    .create_role()
    .role_name(self.role_name.clone())
    .assume_role_policy_document(ROLE_POLICY_DOCUMENT)
    .send()
    .await?;

info!("Created iam role, waiting 15s for it to become active");
tokio::time::sleep(Duration::from_secs(15)).await;

info!("Creating lambda function {}", self.lambda_name);
let _ = self
    .lambda_client
    .create_function()
    .function_name(self.lambda_name.clone())
    .code(code)
    .role(role.role().map(|r| r.arn()).unwrap_or_default())
    .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
    .handler("_unused")
    .send()
    .await
    .map_err(anyhow::Error::from)?;

self.wait_for_function_ready().await?;

self.lambda_client
    .publish_version()
    .function_name(self.lambda_name.clone())
    .send()
    .await?;

Ok(key)
}

/**
 * Upload function code from a path to a zip file.
 * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
 * The easiest way to create such a zip is to use `cargo lambda build -- output-format Zip`.
 */
async fn prepare_function(
    &self,
    zip_file: PathBuf,
    key: Option<String>,
) -> Result<FunctionCode, anyhow::Error> {
    let body = ByteStream::from_path(zip_file).await?

    let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

    info!("Uploading function code to s3://{}", self.bucket, key);
    let _ = self
        .s3_client
        .put_object()
        .bucket(self.bucket.clone())
        .key(key.clone())
        .body(body)
        .send()
        .await?

    Ok(FunctionCode::builder()
        .s3_bucket(self.bucket.clone())
        .s3_key(key)
        .build())
}

• For API details, see CreateFunction in AWS SDK for Rust API reference.

SAP ABAP

SDK for SAP ABAP

⚠️ Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

TRY.

lo_lmd->createfunction(
    iv_functionname = iv_function_name
    iv_runtime = 'python3.9'

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK](https://docs.aws.amazon.com/sdk-for-sap-abap/latest-api/index.html). This topic also includes information about getting started and details about previous SDK versions.

**Delete a Lambda function using an AWS SDK**

The following code examples show how to delete a Lambda function.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

```java
iv_role = iv_role_arn
iv_handler = iv_handler
io_code = io_zip_file
iv_description = 'AWS Lambda code example'
).
MESSAGE 'Lambda function created.' TYPE 'I'.
CATCH /aws1/cx_lmdcodesigningcfgno00.
  MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
CATCH /aws1/cx_lmdcodestorageexcdevex.
  MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
CATCH /aws1/cx_lmdcodeverification00.
  MESSAGE 'Code signature failed one or more validation checks for signature mismatch or expiration.' TYPE 'E'.
CATCH /aws1/cx_lmdinvalidcodesigex.
  MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
CATCH /aws1/cx_lmdinvparamvalueex.
  MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
CATCH /aws1/cx_lmdresourceconflictex.
  MESSAGE 'Resource already exists or another operation is in progress.' TYPE 'E'.
CATCH /aws1/cx_lmdresourcenotfoundex.
  MESSAGE 'The requested resource does not exist.' TYPE 'E'.
CATCH /aws1/cx_lmdserviceexception.
  MESSAGE 'An internal problem was encountered by the AWS Lambda service.' TYPE 'E'.
CATCH /aws1/cx_lmdtoomanyrequestsex.
  MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
ENDTRY.


For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK](https://docs.aws.amazon.com/sdk-for-sap-abap/latest-api/index.html). This topic also includes information about getting started and details about previous SDK versions.

**Delete a Lambda function using an AWS SDK**

The following code examples show how to delete a Lambda function.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:
• Get started with functions

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

/// <summary>
/// Delete an AWS Lambda function.
/// </summary>
/// <param name="functionName">The name of the Lambda function to delete.</param>
/// <returns>A Boolean value that indicates the success of the action.</returns>
public async Task<bool> DeleteFunctionAsync(string functionName)
{
    var request = new DeleteFunctionRequest
    {
        FunctionName = functionName,
    };

    var response = await _lambdaService.DeleteFunctionAsync(request);

    // A return value of NoContent means that the request was processed. In this case, the function was deleted, and the return value is intentionally blank.
}

• For API details, see DeleteFunction in AWS SDK for .NET API Reference.
C++

SDK for C++

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```cpp
Aws::Client::ClientConfiguration clientConfig;
    // Optional: Set to the AWS Region in which the bucket was created
    // (overrides config file).
    // clientConfig.region = "us-east-1";

Aws::Lambda::LambdaClient client(clientConfig);

Aws::Lambda::Model::DeleteFunctionRequest request;
    request.SetFunctionName(LAMBDA_NAME);

Aws::Lambda::Model::DeleteFunctionOutcome outcome = client.DeleteFunction(request);

if (outcome.IsSuccess()) {
    std::cout << "The lambda function was successfully deleted." << std::endl;
} else {
    std::cerr << "Error with Lambda::DeleteFunction. "
                << outcome.GetError().GetMessage() << std::endl;
}
```

- For API details, see [DeleteFunction](https://docs.aws.amazon.com/sdk-for-c++/devguide/AWSLambda-Delete.html) in AWS SDK for C++ API Reference.

CLI

AWS CLI version 2

Example 1: To delete a Lambda function by function name

Delete a function
The following delete-function example deletes the Lambda function named my-function by specifying the function's name.

```bash
aws lambda delete-function \
  --function-name my-function
```

This command produces no output.

**Example 2: To delete a Lambda function by function ARN**

The following delete-function example deletes the Lambda function named my-function by specifying the function's ARN.

```bash
aws lambda delete-function \
```

This command produces no output.

**Example 3: To delete a Lambda function by partial function ARN**

The following delete-function example deletes the Lambda function named my-function by specifying the function's partial ARN.

```bash
aws lambda delete-function \
  --function-name 123456789012:function:my-function
```

This command produces no output.

For more information, see [AWS Lambda Function Configuration](https://docs.aws.amazon.com/lambda/latest/dg/configuration-console.html) in the *AWS Lambda Developer Guide*.

- For API details, see [DeleteFunction](https://docs.aws.amazon.com/cli/latest/reference/lambda/delete-function.html) in *AWS CLI Command Reference*.

Go

**SDK for Go V2**

ℹ️ **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-lambda-code-examples).
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
    LambdaClient *lambda.Client
}

// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(functionName string) {
    _, err := wrapper.LambdaClient.DeleteFunction(context.TODO(),
        &lambda.DeleteFunctionInput{
            FunctionName: aws.String(functionName),
        })
    if err != nil {
        log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
    }
}

• For API details, see DeleteFunction in AWS SDK for Go API Reference.

Java

SDK for Java 2.x

import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lambda.model.DeleteFunctionRequest;
import software.amazon.awssdk.services.lambda.model.LambdaException;

/**

Note

There's more on GitHub. Find the complete example and learn how to set up and run
in the AWS Code Examples Repository.
public class DeleteFunction {
    public static void main(String[] args) {
        final String usage = ":
        Usage:
        <functionName>
        Where:
        functionName - The name of the Lambda function.

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String functionName = args[0];
        Region region = Region.US_EAST_1;
        LambdaClient awsLambda = LambdaClient.builder()
                .region(region)
                .build();

        deleteLambdaFunction(awsLambda, functionName);
        awsLambda.close();
    }

    public static void deleteLambdaFunction(LambdaClient awsLambda, String functionName) {
        try {
            DeleteFunctionRequest request = DeleteFunctionRequest.builder()
                    .functionName(functionName)
                    .build();

            awsLambda.deleteFunction(request);
            System.out.println("The "+functionName +" function was deleted");
        } catch(LambdaException e) {
        }
    }
}
System.err.println(e.getMessage());
System.exit(1);
}
}
}

• For API details, see `DeleteFunction` in *AWS SDK for Java 2.x API Reference*.

**JavaScript**

**SDK for JavaScript (v3)**

```javascript
/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
    const client = new LambdaClient({});
    const command = new DeleteFunctionCommand({ FunctionName: funcName });
    return client.send(command);
};
```

• For API details, see `DeleteFunction` in *AWS SDK for JavaScript API Reference*.

**Kotlin**

**SDK for Kotlin**

```kotlin
/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
    const client = new LambdaClient({});
    const command = new DeleteFunctionCommand({ FunctionName: funcName });
    return client.send(command);
};
```

• For API details, see `DeleteFunction` in *AWS SDK for Kotlin API Reference*.

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-lambda-code-examples).
suspend fun delLambdaFunction(myFunctionName: String) {

    val request = DeleteFunctionRequest {
        functionName = myFunctionName
    }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}

• For API details, see [DeleteFunction](#) in *AWS SDK for Kotlin API reference*.

### PHP

**SDK for PHP**

```php
public function deleteFunction($functionName)
{
    return $this->lambdaClient->deleteFunction(["FunctionName" => $functionName,]);
}
```

• For API details, see [DeleteFunction](#) in *AWS SDK for PHP API Reference*.
Python

SDK for Python (Boto3)

```python
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource

    def delete_function(self, function_name):
        """
        Deletes a Lambda function.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
        raise
```

- For API details, see [DeleteFunction](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

```ruby
# Note
There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).
```
For API details, see [DeleteFunction](#) in *AWS SDK for Ruby API Reference*.

Rust

**SDK for Rust**

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```rust
/** Delete a function and its role, and if possible or necessary, its associated code object and bucket. */
pub async fn delete_function(
    &self,
    location: Option<String>,
) -> Result<DeleteFunctionOutput, anyhow::Error>,
```
```rust
Result<DeleteRoleOutput, anyhow::Error>,
Option<Result<DeleteObjectOutput, anyhow::Error>>,

) {

    info!("Deleting lambda function {self.lambda_name}");
    let delete_function = self
        .lambda_client
        .delete_function()
        .function_name(self.lambda_name.clone())
        .send()
        .await
        .map_err(anyhow::Error::from);

    info!("Deleting iam role {self.role_name}");
    let delete_role = self
        .iam_client
        .delete_role()
        .role_name(self.role_name.clone())
        .send()
        .await
        .map_err(anyhow::Error::from);

    let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
        if let Some(location) = location {
            info!("Deleting object {location}");
            Some(
                self.s3_client
                .delete_object()
                .bucket(self.bucket.clone())
                .key(location)
                .send()
                .await
                .map_err(anyhow::Error::from),
            )
        } else {
            info!(?location, "Skipping delete object");
            None
        }

    (delete_function, delete_role, delete_object)
```

For API details, see **DeleteFunction** in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK](https://docs.aws.amazon.com/lambda/latest/dg/using-with-sap-sdk.html). This topic also includes information about getting started and details about previous SDK versions.

### Get a Lambda function using an AWS SDK

The following code examples show how to get a Lambda function.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:
• Get started with functions

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

    /// <summary>
    /// Gets information about a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function for which to retrieve information.</param>
    /// <returns>Async Task.</returns>
    public async Task<FunctionConfiguration> GetFunctionAsync(string functionName)
    {
        var functionRequest = new GetFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.GetFunctionAsync(functionRequest);
        return response.Configuration;
    }

• For API details, see GetFunction in AWS SDK for .NET API Reference.
C++

SDK for C++

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```cpp
Aws::Client::ClientConfiguration clientConfig;
// Optional: Set to the AWS Region in which the bucket was created
// (overrides config file).
// clientConfig.region = "us-east-1";

Aws::Lambda::LambdaClient client(clientConfig);

Aws::Lambda::Model::GetFunctionRequest request;
request.SetFunctionName(functionName);

Aws::Lambda::Model::GetFunctionOutcome outcome =
    client.GetFunction(request);

if (outcome.IsSuccess()) {
    std::cout << "Function retrieve.\n" <<
    outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable() << std::endl;
}
else {
    std::cerr << "Error with Lambda::GetFunction. " <<
               outcome.GetError().GetMessage() << std::endl;
}
```

- For API details, see [GetFunction](https://docs.aws.amazon.com/sdk-for-cpp/v1/developer-guide/aws-cpp-sdk-lambda.html) in *AWS SDK for C++ API Reference*. 
AWS CLI version 2

To retrieve information about a function

The following `get-function` example displays information about the `my-function` function.

```
aws lambda get-function \
    --function-name  my-function
```

Output:

```
{
    "Concurrency": {
        "ReservedConcurrentExecutions": 100
    },
    "Code": {
        "RepositoryType": "S3",
    },
    "Configuration": {
        "TracingConfig": {
            "Mode": "PassThrough"
        },
        "Version": "$LATEST",
        "CodeSha256": "5tT2qqzYUHoqwR615pZ2dpkn/0J1FrzJm1KidWaaCgk=",
        "FunctionName": "my-function",
        "VpcConfig": {
            "SubnetIds": [],
            "VpcId": "",
            "SecurityGroupIds": []
        },
        "MemorySize": 128,
        "RevisionId": "28f0fb31-5c5c-43d3-8955-03e76c5c1075",
        "CodeSize": 304,
        "Handler": "index.handler",
        "Role": "arn:aws:iam::123456789012:role/service-role/helloWorldPython-role-uy319qyq"
}
```
For more information, see [AWS Lambda Function Configuration](https://docs.aws.amazon.com/lambda/latest/dg/configuration-handler.html) in the *AWS Lambda Developer Guide*.

- For API details, see [GetFunction](https://docs.aws.amazon.com/cli/latest/reference/lambda/get-function.html) in *AWS CLI Command Reference*.

**Go**

**SDK for Go V2**

```go
// FunctionWrapper encapsulates function actions used in the examples. // It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
    LambdaClient *lambda.Client
}

// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(functionName string) types.State {
    var state types.State
    funcOutput, err := wrapper.LambdaClient.GetFunction(context.TODO(),
        &lambda.GetFunctionInput{
            FunctionName: aws.String(functionName),
        })
    if err != nil {
        log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
    } else {
        state = funcOutput.Configuration.State
    }
}
```

*Note*  
There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-code-examples).
• For API details, see `GetFunction` in *AWS SDK for Go API Reference*.

JavaScript

**SDK for JavaScript (v3)**

```javascript
const getFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new GetFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

• For API details, see `GetFunction` in *AWS SDK for JavaScript API Reference*.

PHP

**SDK for PHP**

```php
public function getFunction($functionName)
```

• For API details, see `GetFunction` in *AWS SDK for JavaScript API Reference*.

> **Note**
> There's more on GitHub. Find the complete example and learn how to set up and run in the *AWS Code Examples Repository*.
```
return $this->lambdaClient->getFunction(
    'FunctionName' => $functionName,
);}
```

- For API details, see [GetFunction](https://docs.aws.amazon.com/lambda/latest/dg/lambda-api-arguments.html) in *AWS SDK for PHP API Reference*.

**Python**

**SDK for Python (Boto3)**

```python
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource

    def get_function(self, function_name):
        
        Gets data about a Lambda function.

        :param function_name: The name of the function.
        :return: The function data.
        
        response = None
        try:
            response =
            self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response['Error']['Code'] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
                logger.error("Couldn't get function %s. Here's why: %s",

Get a function 1794
function_name,
err.response["Error"]['Code'],
err.response["Error"]['Message'],
)
raise
return response

- For API details, see `GetFunction` in *AWS SDK for Python (Boto3) API Reference*.

**Ruby**

**SDK for Ruby**

```ruby
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Gets data about a Lambda function.
  #
  # @param function_name: The name of the function.
  # @return response: The function data, or nil if no such function exists.
  def get_function(function_name)
    @lambda_client.get_function(
      {
        function_name: function_name
      }
    )
    rescue Aws::Lambda::Errors::ResourceNotFoundException => e
      @logger.debug("Could not find function: #{function_name}:
                      #{e.message}"
    end
  end
end
```

*Note*

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-code-examples).
- For API details, see `GetFunction` in *AWS SDK for Ruby API Reference*.

### Rust

**SDK for Rust**

```rust
/** Get the Lambda function with this Manager's name. */
pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error> {
    info!("Getting lambda function");
    self.lambda_client
        .get_function()
        .function_name(self.lambda_name.clone())
        .send()
        .await
        .map_err(anyhow::Error::from)
}
```

- For API details, see `GetFunction` in *AWS SDK for Rust API reference*.

### SAP ABAP

**SDK for SAP ABAP**

```abap
Note
There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://aws.amazon.com/code-examples/).
```
TRY.
  oo_result = lo_lmd->getfunction( iv_functionname = iv_function_name ).
" oo_result is returned for testing purposes."
  MESSAGE 'Lambda function information retrieved.' TYPE 'I'.
CATCH /aws1/cx_lmdinvpamvalueex.
  MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
CATCH /aws1/cx_lmdserviceexception.
  MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
  TYPE 'E'.
CATCH /aws1/cx_lmdtoomanyrequestsex.
  MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
ENDTRY.

- For API details, see `GetFunction` in `AWS SDK for SAP ABAP API reference`.

For a complete list of AWS SDK developer guides and code examples, see `Using Lambda with an AWS SDK`. This topic also includes information about getting started and details about previous SDK versions.

**Invoke a Lambda function using an AWS SDK**

The following code examples show how to invoke a Lambda function.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- [Get started with functions](#)

**.NET**

**AWS SDK for .NET**

<i>Note</i>

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).
```csharp
/// Invoke a Lambda function.
/// </summary>
/// <param name="functionName">The name of the Lambda function to invoke.</param>
/// <param name="parameters">The parameter values that will be passed to the function.</param>
/// <returns>A System Threading Task.</returns>
public async Task<string> InvokeFunctionAsync(
    string functionName,
    string parameters)
{
    var payload = parameters;
    var request = new InvokeRequest
        {
            FunctionName = functionName,
            Payload = payload,
        };

    var response = await _lambdaService.InvokeAsync(request);
    MemoryStream stream = response.Payload;
    string returnValue =
        System.Text.Encoding.UTF8.GetString(stream.ToArray());
    return returnValue;
}
```

- For API details, see Invoke in AWS SDK for .NET API Reference.

C++

SDK for C++

```
Aws::Client::ClientConfiguration clientConfig;
// Optional: Set to the AWS Region in which the bucket was created (overrides config file).
```

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
For API details, see Invoke in AWS SDK for C++ API Reference.

CLI

AWS CLI version 2

Example 1: To invoke a Lambda function synchronously

The following invoke example invokes the my-function function synchronously. The cli-binary-format option is required if you're using AWS CLI version 2. For more information, see AWS CLI supported global command line options in the AWS Command Line Interface User Guide.
aws lambda invoke \n   --function-name my-function \n   --cli-binary-format raw-in-base64-out \n   --payload '{ "name": "Bob" }' \nresponse.json

Output:

{  
   "ExecutedVersion": "$LATEST",  
   "StatusCode": 200
}

For more information, see [Synchronous Invocation](https://docs.aws.amazon.com/lambda/latest/dg/synchronous-invocation.html) in the *AWS Lambda Developer Guide*.

**Example 2: To invoke a Lambda function asynchronously**

The following `invoke` example invokes the `my-function` function asynchronously. The `cli-binary-format` option is required if you're using AWS CLI version 2. For more information, see [AWS CLI supported global command line options](https://docs.aws.amazon.com/cli/latest/reference/index.html) in the *AWS Command Line Interface User Guide*.

aws lambda invoke \n   --function-name my-function \n   --invocation-type Event \n   --cli-binary-format raw-in-base64-out \n   --payload '{ "name": "Bob" }' \nresponse.json

Output:

{  
   "StatusCode": 202
}

For more information, see [Asynchronous Invocation](https://docs.aws.amazon.com/lambda/latest/dg/asynchronous-invocation.html) in the *AWS Lambda Developer Guide*.

- For API details, see [Invoke](https://docs.aws.amazon.com/lambda/latest/dg/lambda-api-invoke.html) in *AWS CLI Command Reference*. 
Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```go
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
    LambdaClient *lambda.Client
}

// Invoke invokes the Lambda function specified by functionName, passing the parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which tells
// Lambda to include the last few log lines in the returned result.
func (wrapper FunctionWrapper) Invoke(functionName string, parameters any, getLog bool) *lambda.InvokeOutput {
    logType := types.LogTypeNone
    if getLog {
        logType = types.LogTypeTail
    }
    payload, err := json.Marshal(parameters)
    if err != nil {
        log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
    }
    invokeOutput, err := wrapper.LambdaClient.Invoke(context.TODO(),
        &lambda.InvokeInput{
            FunctionName: aws.String(functionName),
            LogType:      logType,
            Payload:      payload,
        })
    if err != nil {
        log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
    }
    return invokeOutput
}
```
For API details, see Invoke in AWS SDK for Go API Reference.

Java

SDK for Java 2.x

import org.json.JSONObject;
import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lambda.model.InvokeRequest;
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.services.lambda.model.InvokeResponse;
import software.amazon.awssdk.services.lambda.model.LambdaException;

public class LambdaInvoke {

  /*
   * you can retrieve the value by looking at the function in the AWS Console
   * Also, set up your development environment, including your credentials.
   * For information, see this documentation topic:
   * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
   */

  {}
public static void main(String[] args) {
    final String usage = ""
    Usage:
        <functionName>\s
    Where:
        functionName - The name of the Lambda function\s
        "";
    if (args.length != 1) {
        System.out.println(usage);
        System.exit(1);
    }
    String functionName = args[0];
    Region region = Region.US_WEST_2;
    LambdaClient awsLambda = LambdaClient.builder()
        .region(region)
        .build();
    invokeFunction(awsLambda, functionName);
    awsLambda.close();
}

public static void invokeFunction(LambdaClient awsLambda, String
    functionName) {
    InvokeResponse res = null;
    try {
        // Need a SdkBytes instance for the payload.
        JSONObject jsonObj = new JSONObject();
        jsonObj.put("inputValue", "2000");
        String json = jsonObj.toString();
        SdkBytes payload = SdkBytes.fromUtf8String(json);
        // Setup an InvokeRequest.
        InvokeRequest request = InvokeRequest.builder()
            .functionName(functionName)
            .payload(payload)
            .build();
        res = awsLambda.invoke(request);
        String value = res.payload().asUtf8String();
    }
System.out.println(value);
}
}
}

For API details, see Invoke in AWS SDK for Java 2.x API Reference.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

const invoke = async (funcName, payload) => {
    const client = new LambdaClient({});
    const command = new InvokeCommand({
        FunctionName: funcName,
        Payload: JSON.stringify(payload),
        LogType: LogType.Tail,
    });

    const { Payload, LogResult } = await client.send(command);
    const result = Buffer.from(Payload).toString();
    const logs = Buffer.from(LogResult, "base64").toString();
    return { logs, result };};

For API details, see Invoke in AWS SDK for JavaScript API Reference.
Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```kotlin
suspend fun invokeFunction(functionNameVal: String) {
    val json = """{"inputValue":"1000"""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request = InvokeRequest {
        functionName = functionNameVal
        logType = LogType.Tail
        payload = byteArray
    }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
        println("${res.payload?.toString(Charsets.UTF_8)}")
        println("The log result is ${res.logResult}")
    }
}
```

- For API details, see Invoke in AWS SDK for Kotlin API reference.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
public function invoke($functionName, $params, $logType = 'None')
{
    return $this->lambdaClient->invoke([  
        'FunctionName' => $functionName,
        'Payload' => json_encode($params),
        'LogType' => $logType,
    ]);   
}

• For API details, see Invoke in AWS SDK for PHP API Reference.

Python

SDK for Python (Boto3)

class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource

    def invoke_function(self, function_name, function_params, get_log=False):
        """
        Invokes a Lambda function.
        """
        :param function_name: The name of the function to invoke.
        :param function_params: The parameters of the function as a dict. This dict
        is serialized to JSON before it is sent to Lambda.
        :param get_log: When true, the last 4 KB of the execution log are included in
        the response.
        :return: The response from the function invocation.
```python
try:
    response = self.lambda_client.invoke(
        FunctionName=function_name,
        Payload=json.dumps(function_params),
        LogType="Tail" if get_log else "None",
    )
    logger.info("Invoked function %s.", function_name)
except ClientError:
    logger.exception("Couldn't invoke function %s.", function_name)
    raise
return response
```

- For API details, see `Invoke` in *AWS SDK for Python (Boto3) API Reference*.

**Ruby**

**SDK for Ruby**

```ruby
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Invokes a Lambda function.
  # @param function_name [String] The name of the function to invoke.
  # @param payload [nil] Payload containing runtime parameters.
  # @return [Object] The response from the function invocation.
  def invoke_function(function_name, payload = nil)
    params = { function_name: function_name}
```

*Invoke a function*
For API details, see Invoke in AWS SDK for Ruby API Reference.

Rust

SDK for Rust

```rust
/** Invoke the lambda function using calculator InvokeArgs. */
pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput, anyhow::Error> {
    info!(?args, "Invoking {}", self.lambda_name);
    let payload = serde_json::to_string(&args)?;
    debug!(?payload, "Sending payload");
    self.lambda_client
        .invoke()
        .function_name(self.lambda_name.clone())
        .payload(Blob::new(payload))
        .send()
        .await
        .map_err(anyhow::Error::from)
}

fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
        info!("Could not extract payload")
    }
}
```

Invoke a function
if let Some(logs) = invoke.log_result() {
    debug!(?logs, "Invoked function logs")
} else {
    debug!("Invoked function had no logs")
} 

• For API details, see Invoke in AWS SDK for Rust API reference.

SAP ABAP

SDK for SAP ABAP

TRY.
    DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
        '{
            "action": "increment", 
            "number": 10
        }
    ).
    oo_result = lo_lmd->invoke(                   " oo_result is returned for
        iv_functionname = iv_function_name
        iv_payload = lv_json
    ).
    MESSAGE 'Lambda function invoked.' TYPE 'I'.
    CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
    CATCH /aws1/cx_lmdinvrequestcontextex.
        MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
    CATCH /aws1/cx_lmdinvalidzipfileex.
        MESSAGE 'The deployment package could not be unzipped.' TYPE 'E'.
    CATCH /aws1/cx_lmdrequesttoolargeex.
        MESSAGE 'Invoke request body JSON input limit was exceeded by the request
payload.' TYPE 'E'.
CATCH /aws1/cx_lmdresourceconflictex.
  MESSAGE 'Resource already exists or another operation is in progress.' TYPE 'E'.
CATCH /aws1/cx_lmdresourcenotfoundex.
  MESSAGE 'The requested resource does not exist.' TYPE 'E'.
CATCH /aws1/cx_lmdserviceexception.
  MESSAGE 'An internal problem was encountered by the AWS Lambda service.' TYPE 'E'.
CATCH /aws1/cx_lmdtoomanyrequestsex.
  MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
CATCH /aws1/cx_lmdunsuppedmediatyp00.
  MESSAGE 'Invoke request body does not have JSON as its content type.' TYPE 'E'.
ENDTRY.

• For API details, see Invoke in AWS SDK for SAP ABAP API reference.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

List Lambda functions using an AWS SDK

The following code examples show how to list Lambda functions.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

• Get started with functions

.NET

AWS SDK for .NET

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
/// <summary>
/// Get a list of Lambda functions.
/// </summary>
/// <returns>A list of FunctionConfiguration objects.</returns>
public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
{
    var functionList = new List<FunctionConfiguration>();

    var functionPaginator =
        _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
    await foreach (var function in functionPaginator.Functions)
    {
        functionList.Add(function);
    }

    return functionList;
}

- For API details, see ListFunctions in AWS SDK for .NET API Reference.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Aws::Client::ClientConfiguration clientConfig;
// Optional: Set to the AWS Region in which the bucket was created
// (overrides config file).
// clientConfig.region = "us-east-1";

Aws::Lambda::LambdaClient client(clientConfig);

std::vector<Aws::String> functions;
Aws::String marker;
do {
    Aws::Lambda::Model::ListFunctionsRequest request;
    if (!marker.empty()) {
        request.SetMarker(marker);
    }

    Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(request);

    if (outcome.IsSuccess()) {
        const Aws::Lambda::Model::ListFunctionsResult &result = outcome.GetResult();
        std::cout << result.GetFunctions().size() << " lambda functions were retrieved." << std::endl;

        for (const Aws::Lambda::Model::FunctionConfiguration &functionConfiguration: result.GetFunctions()) {
            functions.push_back(functionConfiguration.GetFunctionName());
            std::cout << functions.size() << " " << functionConfiguration.GetDescription() << std::endl;
            std::cout << " " << Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(functionConfiguration.GetRuntime()) << ": " << functionConfiguration.GetHandler() << std::endl;
        }
        marker = result.GetNextMarker();
    } else {
        std::cerr << "Error with Lambda::ListFunctions. " << outcome.GetError().GetMessage() << std::endl;
    }
} while (!marker.empty());

• For API details, see ListFunctions in AWS SDK for C++ API Reference.
AWS CLI version 2

To retrieve a list of Lambda functions

The following `list-functions` example displays a list of all of the functions for the current user.

```
aws lambda list-functions
```

Output:

```
{
    "Functions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "dBG9m8SGdm1Ejw/JYX1hhvCrAv5TxvXsbL/RMrfT/I=",
            "FunctionName": "helloworld",
            "MemorySize": 128,
            "RevisionId": "1718e831-badf-4253-9518-d0644210af7b",
            "CodeSize": 294,
            "Handler": "helloworld.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-zgur6bf4",
            "Timeout": 3,
            "LastModified": "2019-09-23T18:32:33.857+0000",
            "Runtime": "nodejs10.x",
            "Description": ""
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/1TxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
```
"SubnetIds": [],
"VpcId": "",
"SecurityGroupIds": []
},
"MemorySize": 256,
"RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668",
"CodeSize": 266,
"Handler": "index.handler",
"Role": "arn:aws:iam::123456789012:role/service-role/helloWorldPython-role-uy319qqq",
"Timeout": 3,
"LastModified": "2019-10-01T16:47:28.490+0000",
"Runtime": "nodejs10.x",
"Description": ""
},
{
"Layers": [
{
"CodeSize": 41784542,
},
{
"CodeSize": 4121,
}
],
"TracingConfig": {
"Mode": "PassThrough"
},
"Version": "$LATEST",
"CodeSha256": "ZQukCqxtkFgyF2cU41Avj99TKQ/hNihPtDtRcc08mI=",
"FunctionName": "my-python-function",
"VpcConfig": {
"SubnetIds": [],
"VpcId": "",
"SecurityGroupIds": []
},
"MemorySize": 128,
"RevisionId": "80b4ebac-acf7-4ea8-919a-e874c213707d",
"CodeSize": 299,
  "Handler": "lambda_function.lambda_handler",
  "Role": "arn:aws:iam::123456789012:role/service-role/my-python-function-role-z5g7dr6n",
  "Timeout": 3,
  "LastModified": "2019-10-01T19:40:41.643+0000",
  "Runtime": "python3.7",
  "Description": ""
}
]
}

For more information, see AWS Lambda Function Configuration in the AWS Lambda Developer Guide.

- For API details, see ListFunctions in AWS CLI Command Reference.

Go

**SDK for Go V2**

```go
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
  LambdaClient *lambda.Client
}

// ListFunctions lists up to maxItems functions for the account. This function uses a
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(maxItems int) []types.FunctionConfiguration {
```

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
var functions []types.FunctionConfiguration
paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
&lambda.ListFunctionsInput{
    MaxItems: aws.Int32(int32(maxItems)),
})
for paginator.HasMorePages() && len(functions) < maxItems {
    pageOutput, err := paginator.NextPage(context.TODO())
    if err != nil {
        log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
    }
    functions = append(functions, pageOutput.Functions...)
}
return functions
}

- For API details, see ListFunctions in AWS SDK for Go API Reference.

JavaScript

SDK for JavaScript (v3)

const listFunctions = () => {
    const client = new LambdaClient({});
    const command = new ListFunctionsCommand({});

    return client.send(command);
};

- For API details, see ListFunctions in AWS SDK for JavaScript API Reference.
Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```php
public function listFunctions($maxItems = 50, $marker = null) {
    if (is_null($marker)) {
        return $this->lambdaClient->listFunctions([ 'MaxItems' => $maxItems,
    ]);}

    return $this->lambdaClient->listFunctions([ 'Marker' => $marker,
        'MaxItems' => $maxItems,
    ]);}
```

- For API details, see ListFunctions in AWS SDK for PHP API Reference.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
```

List functions
def list_functions(self):
    """
    Lists the Lambda functions for the current account.
    """
    try:
        func_paginator = self.lambda_client.get_paginator("list_functions")
        for func_page in func_paginator.paginate():
            for func in func_page["Functions"]:  
                print(func["FunctionName"])
                desc = func.get("Description")
                if desc:
                    print(f"\t{desc}"
                print(f"\t{func['Runtime']}\t{func['Handler']}")
    except ClientError as err:
        logger.error("Couldn't list functions. Here's why: %s: %s",
                     err.response["Error"]['Code'],
                     err.response["Error"]['Message'],
             )
        raise

• For API details, see ListFunctions in AWS SDK for Python (Boto3) API Reference.

Ruby

SDK for Ruby

蜻 " Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

class LambdaWrapper
    attr_accessor :lambda_client

    def initialize
@lambda_client = Aws::Lambda::Client.new
@logger = Logger.new($stdout)
@logger.level = Logger::WARN
end

# Lists the Lambda functions for the current account.
def list_functions
  functions = []
  @lambda_client.list_functions.each do |response|
    response["functions"].each do |function|
      functions.append(function["function_name"])
    end
  end
  functions
rescue Aws::Lambda::Errors::ServiceException => e
  @logger.error("There was an error executing #{function_name}:
  #{e.message}")
end

- For API details, see ListFunctions in AWS SDK for Ruby API Reference.

Rust

SDK for Rust

```rust
/** List all Lambda functions in the current Region. */
pub async fn list_functions(&self) -> Result<ListFunctionsOutput, anyhow::Error> {
    info!("Listing lambda functions");
    self.lambda_client
        .list_functions()
        .send()
        .await
        .map_err(anyhow::Error::from)
}
```

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
For API details, see ListFunctions in AWS SDK for Rust API reference.

SAP ABAP

SDK for SAP ABAP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

TRY.

    oo_result = lo_lmd->listfunctions( ).    " oo_result is returned for testing purposes. "
    DATA(lt_functions) = oo_result->get_functions( ).
    MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
    CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
    CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.' TYPE 'E'.
    CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
ENDTRY.

For API details, see ListFunctions in AWS SDK for SAP ABAP API reference.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Update Lambda function code using an AWS SDK

The following code examples show how to update Lambda function code.
Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- **Get started with functions**

.NET

**AWS SDK for .NET**

```csharp
/// <summary>
/// Update an existing Lambda function.
/// </summary>
/// <param name="functionName">The name of the Lambda function to update.</param>
/// <param name="bucketName">The bucket where the zip file containing the Lambda function code is stored.</param>
/// <param name="key">The key name of the source code file.</param>
/// <returns>Async Task.</returns>
public async Task UpdateFunctionCodeAsync(
    string functionName,
    string bucketName,
    string key)
{
    var functionCodeRequest = new UpdateFunctionCodeRequest
    {
        FunctionName = functionName,
        Publish = true,
        S3Bucket = bucketName,
        S3Key = key,
    };

    var response = await _lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
    Console.WriteLine($"The Function was last modified at {response.LastModified}";
```

**Update function code**

1821
• For API details, see [UpdateFunctionCode](#) in *AWS SDK for .NET API Reference.*

### C++

#### SDK for C++

```cpp
Aws::Client::ClientConfiguration clientConfig;
// Optional: Set to the AWS Region in which the bucket was created
// (overrides config file).
// clientConfig.region = "us-east-1";

Aws::Lambda::LambdaClient client(clientConfig);

Aws::Lambda::Model::UpdateFunctionCodeRequest request;
request.SetFunctionName(LAMBDA_NAME);
std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                       std::ios_base::in | std::ios_base::binary);
if (!ifstream.is_open()) {
    std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." << std::endl;

    #if USE_CPP_LAMBDA_FUNCTION
    std::cerr
        << "The cpp Lambda function must be built following the instructions in the cpp_lambda/README.md file."
        << std::endl;
    #endif
    deleteLambdaFunction(client);
    deleteIamRole(clientConfig);
    return false;
}
```
Aws::StringStream buffer;
buffer << ifstream.rdbuf();
request.SetZipFile(
    Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                            buffer.str().length()));
request.SetPublish(true);

Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
    client.UpdateFunctionCode(
        request);

if (outcome.IsSuccess()) {
    std::cout << "The lambda code was successfully updated." <<
    std::endl;
} else {
    std::cerr << "Error with Lambda::UpdateFunctionCode. "
              << outcome.GetError().GetMessage() 
              << std::endl;
}

• For API details, see UpdateFunctionCode in AWS SDK for C++ API Reference.

CLI

AWS CLI version 2

To update the code of a Lambda function

The following update-function-code example replaces the code of the unpublished
($LATEST) version of the my-function function with the contents of the specified zip file.

aws lambda update-function-code \
  --function-name my-function \
  --zip-file fileb://my-function.zip

Output:

{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",}
For more information, see [AWS Lambda Function Configuration](https://docs.aws.amazon.com/lambda/latest/dg/configuration-goinger.html) in the [AWS Lambda Developer Guide](https://docs.aws.amazon.com/lambda/latest/dg/).


## Go

### SDK for Go V2

```go
// FunctionWrapper encapsulates function actions used in the examples.  
// It contains an AWS Lambda service client that is used to perform user actions.

type FunctionWrapper struct {
    LambdaClient *lambda.Client
}
```

---

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-lambda-go).
// UpdateFunctionCode updates the code for the Lambda function specified by
// functionName. // The existing code for the Lambda function is entirely replaced by the code in
// the // zipPackage buffer. After the update action is called, a
// lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(functionName string, zipPackage
*bytes.Buffer) types.State {
    var state types.State
    _, err := wrapper.LambdaClient.UpdateFunctionCode(context.TODO(),
&lambda.UpdateFunctionCodeInput{
    FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
})
    if err != nil {
        log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
functionName, err)
    } else {
        waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
        funcOutput, err := waiter.WaitForOutput(context.TODO(),
&lambda.GetFunctionInput{
    FunctionName: aws.String(functionName)}, 1*time.Minute)
        if err != nil {
            log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
functionName, err)
        } else {
            state = funcOutput.Configuration.State
        }
    }
    return state
}

- For API details, see **UpdateFunctionCode** in *AWS SDK for Go API Reference*. 
JavaScript

SDK for JavaScript (v3)

```javascript
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand(
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  );

  return client.send(command);
};
```

- For API details, see [UpdateFunctionCode](https://aws.amazon.com/documentation/lambda/api-ref/) in [AWS SDK for JavaScript API Reference](https://aws.amazon.com/documentation/lambda/api-ref/).

PHP

SDK for PHP

```php
public function updateFunctionCode($functionName, $s3Bucket, $s3Key)
{
```

- For API details, see [UpdateFunctionCode](https://aws.amazon.com/documentation/lambda/api-ref/) in [AWS SDK for PHP API Reference](https://aws.amazon.com/documentation/lambda/api-ref/).
For API details, see `UpdateFunctionCode` in AWS SDK for PHP API Reference.

Python

SDK for Python (Boto3)

```python
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource

    def update_function_code(self, function_name, deployment_package):
        """
        Updates the code for a Lambda function by submitting a .zip archive that contains
        the code for the function.

        :param function_name: The name of the function to update.
        :param deployment_package: The function code to update, packaged as bytes in .zip format.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_code(
                FunctionName=function_name, ZipFile=deployment_package
            )
```

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
except ClientError as err:
    logger.error(
        "Couldn't update function %s. Here's why: %s: %s",
        function_name,
        err.response["Error"]['Code'],
        err.response["Error"]['Message'],
    )
    raise
else:
    return response

• For API details, see `UpdateFunctionCode` in *AWS SDK for Python (Boto3) API Reference*.

Ruby

**SDK for Ruby**

```ruby
# Updates the code for a Lambda function by submitting a .zip archive that
# contains
# the code for the function.

# @param function_name: The name of the function to update.
# @param deployment_package: The function code to update, packaged as bytes in
# .zip format.
# @return: Data about the update, including the status.
```

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run
in the *AWS Code Examples Repository*.
def update_function_code(function_name, deployment_package)
    @lambda_client.update_function_code(
        function_name: function_name,
        zip_file: deployment_package
    )
    @lambda_client.wait_until(:function_updated_v2, { function_name:
        function_name}) do |w|
        w.max_attempts = 5
        w.delay = 5
    end
    rescue Aws::Lambda::Errors::ServiceException => e
        @logger.error("There was an error updating function code for:
        #{function_name}:
        #{e.message}"
        )
        nil
    rescue Aws::Waiters::Errors::WaiterFailed => e
        @logger.error("Failed waiting for #{function_name} to update:
        #{e.message}"
        )
    end

• For API details, see UpdateFunctionCode in AWS SDK for Ruby API Reference.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

/** Given a Path to a zip file, update the function's code and wait for the update to finish. */
pub async fn update_function_code(
    &self,
    zip_file: PathBuf,
    key: String,
) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
    let function_code = self.prepare_function(zip_file, Some(key)).await?
    info!("Updating code for ", self.lambda_name);
let update = self
    .lambda_client
    .update_function_code()
    .function_name(self.lambda_name.clone())
    .s3_bucket(self.bucket.clone())
    .s3_key(function_code.s3_key().unwrap().to_string())
    .send()
    .await
    .map_err(anyhow::Error::from)?;

self.wait_for_function_ready().await?;

Ok(update)
}
• For API details, see UpdateFunctionCode in AWS SDK for Rust API reference.

SAP ABAP

SDK for SAP ABAP

TRY.

    oo_result = lo_lmd->updatefunctioncode( " oo_result is returned for testing purposes. "
        iv_functionname = iv_function_name
        iv_zipfile = io_zip_file
    ).

    MESSAGE 'Lambda function code updated.' TYPE 'I'.
CATCH /aws1/cx_lmdcodesigningcfgno00.
    MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
CATCH /aws1/cx_lmdcodestorageexcdex.
    MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
CATCH /aws1/cx_lmdcodeverification00.
    MESSAGE 'Code signature failed one or more validation checks for signature mismatch or expiration.' TYPE 'E'.
CATCH /aws1/cx_lmdinvaliddesignex.
    MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
CATCH /aws1/cx_lmdinvalidparamvalueex.
    MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
CATCH /aws1/cx_lmdresourceconflictex.
    MESSAGE 'Resource already exists or another operation is in progress.' TYPE 'E'.
CATCH /aws1/cx_lmdresourcenotfoundex.
    MESSAGE 'The requested resource does not exist.' TYPE 'E'.
CATCH /aws1/cx_lmdserviceexception.
    MESSAGE 'An internal problem was encountered by the AWS Lambda service.' TYPE 'E'.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
CATCH /aws1/cx_lmdtomanyrequestsex.
  MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
ENDTRY.

- For API details, see UpdateFunctionCode in AWS SDK for SAP ABAP API reference.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

**Update Lambda function configuration using an AWS SDK**

The following code examples show how to update Lambda function configuration.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with functions

**.NET**

**AWS SDK for .NET**

Note
---

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```csharp
/// <summary>
/// Update the code of a Lambda function.
/// </summary>
/// <param name="functionName">The name of the function to update.</param>
/// <param name="functionHandler">The code that performs the function's actions.</param>
/// <param name="environmentVariables">A dictionary of environment variables.</param>
/// <returns>A Boolean value indicating the success of the action.</returns>
public async Task<bool> UpdateFunctionConfigurationAsync(
```
string functionName,
string functionHandler,
Dictionary<string, string> environmentVariables)
{
    var request = new UpdateFunctionConfigurationRequest
    {
        Handler = functionHandler,
        FunctionName = functionName,
        Environment = new Amazon.Lambda.Model.Environment { Variables =
            environmentVariables },
    }

    var response = await _lambdaService.UpdateFunctionConfigurationAsync(request);

    Console.WriteLine(response.LastModified);

}

• For API details, see UpdateFunctionConfiguration in AWS SDK for .NET API Reference.

C++

SDK for C++

Note
There's more on GitHub. Find the complete example and learn how to set up and run
in the AWS Code Examples Repository.

Aws::Client::ClientConfiguration clientConfig;
// Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
// clientConfig.region = "us-east-1";

Aws::Lambda::LambdaClient client(clientConfig);

Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
request.SetFunctionName(LAMBDA_NAME);
Aws::Lambda::Model::Environment environment;
environment.AddVariables("LOG_LEVEL", "DEBUG");
request.SetEnvironment(environment);

Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
client.UpdateFunctionConfiguration(
    request);

if (outcome.IsSuccess()) {
    std::cout << "The lambda configuration was successfully updated."
    << std::endl;
    break;
}
else {
    std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
    << outcome.GetError().GetMessage()
    << std::endl;
}

• For API details, see UpdateFunctionConfiguration in AWS SDK for C++ API Reference.

CLI

AWS CLI version 2

To modify the configuration of a function

The following update-function-configuration example modifies the memory size to be 256 MB for the unpublished ($LATEST) version of the my-function function.

```
aws lambda update-function-configuration \
    --function-name my-function \ 
    --memory-size 256
```

Output:

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
```
"RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
"MemorySize": 256,
"Version": "$LATEST",
"Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-uy3l9qyq",
"Timeout": 3,
"Runtime": "nodejs10.x",
"TracingConfig": {
    "Mode": "PassThrough"
},
"CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
"Description": "",
"VpcConfig": {
    "SubnetIds": [],
    "VpcId": "",
    "SecurityGroupIds": []
},
"CodeSize": 304,
"Handler": "index.handler"
}

For more information, see [AWS Lambda Function Configuration](#) in the [AWS Lambda Developer Guide](#).

- For API details, see [UpdateFunctionConfiguration](#) in [AWS CLI Command Reference](#).

Go

SDK for Go V2

ℹ️ Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```go
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
    LambdaClient *lambda.Client
```
// UpdateFunctionConfiguration updates a map of environment variables configured for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(functionName string,
    envVars map[string]string) {
    _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(context.TODO(),
        &lambda.UpdateFunctionConfigurationInput{
            FunctionName: aws.String(functionName),
            Environment: &types.Environment{Variables: envVars},
        })
    if err != nil {
        log.Panicf("Couldn't update configuration for %v. Here's why: %v",
            functionName, err)
    }
}

- For API details, see UpdateFunctionConfiguration in AWS SDK for Go API Reference.

JavaScript

SDK for JavaScript (v3)

```javascript
const updateFunctionConfiguration = (funcName) => {
    const client = new LambdaClient({});
    const config = readFileSync(`${dirname}../functions/config.json`).toString();
    const command = new UpdateFunctionConfigurationCommand({
        ...JSON.parse(config),
        FunctionName: funcName,
    });
    return client.send(command);
};
```

° Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
• For API details, see [UpdateFunctionConfiguration](https://docs.aws.amazon.com/sdk-for-php/api/v3/service-reference/lambda.html#UpdateFunctionConfiguration) in [AWS SDK for PHP API Reference](https).

**PHP**

**SDK for PHP**

```php
public function updateFunctionConfiguration($functionName, $handler, $environment = '') {
    return $this->lambdaClient->updateFunctionConfiguration(["FunctionName" => $functionName,
                                                              "Handler" => "$handler.lambda_handler",
                                                              "Environment" => $environment,]);
}
```

• For API details, see [UpdateFunctionConfiguration](https://docs.aws.amazon.com/sdk-for-php/api/v3/service-reference/lambda.html#UpdateFunctionConfiguration) in [AWS SDK for PHP API Reference](https).

**Python**

**SDK for Python (Boto3)**

```python
class LambdaWrapper:
```

[Note]
There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https).
def __init__(self, lambda_client, iam_resource):
    self.lambda_client = lambda_client
    self.iam_resource = iam_resource

def update_function_configuration(self, function_name, env_vars):
    
    Updates the environment variables for a Lambda function.

    :param function_name: The name of the function to update.
    :param env_vars: A dict of environment variables to update.
    :return: Data about the update, including the status.
    
    try:
        response = self.lambda_client.update_function_configuration(
            FunctionName=function_name, Environment={"Variables": env_vars}
        )
    except ClientError as err:
        logger.error(
            "Couldn't update function configuration %s. Here's why: %s: %s",
            function_name,
            err.response["Error"]['Code'],
            err.response["Error"]['Message'],
        )
        raise
    else:
        return response

• For API details, see UpdateFunctionConfiguration in AWS SDK for Python (Boto3) API Reference.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Updates the environment variables for a Lambda function.
  # @param function_name: The name of the function to update.
  # @param log_level: The log level of the function.
  # @return: Data about the update, including the status.
  def update_function_configuration(function_name, log_level)
    @lambda_client.update_function_configuration(
      function_name: function_name,
      environment: {
        variables: {
          "LOG_LEVEL" => log_level
        }
      }
    )
    @lambda_client.wait_until(:function_updated_v2, { function_name: function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
    rescue Aws::Lambda::Errors::ServiceException => e
      @logger.error("There was an error updating configurations for 
      #{function_name}:
      #{e.message}")
    rescue Aws::Waiters::Errors::WaiterFailed => e
      @logger.error("Failed waiting for #{function_name} to activate:
      #{e.message}"
    end
  end

  • For API details, see UpdateFunctionConfiguration in AWS SDK for Ruby API Reference.
Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```rust
/** Update the environment for a function. */
pub async fn update_function_configuration(
    &self,
    environment: Environment,
) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
    info!(
        ?environment,
        "Updating environment for {}", self.lambda_name
    );
    let updated = self
        .lambda_client
        .update_function_configuration()
        .function_name(self.lambda_name.clone())
        .environment(environment)
        .send()
        .await
        .map_err(anyhow::Error::from)?;
    self.wait_for_function_ready().await?
    Ok(updated)
}
```

- For API details, see [UpdateFunctionConfiguration](https://aws.amazon.com/) in AWS SDK for Rust API reference.
TRY.
    oo_result = lo_lmd->updatefunctionconfiguration(   " oo_result is
        iv_functionname = iv_function_name
        iv_runtime = iv_runtime
        iv_description = 'Updated Lambda function'
        iv_memorysize = iv_memory_size
    )

    MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
CATCH /aws1/cx_lmdcodesigningcfgno00.
    MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
CATCH /aws1/cx_lmdcodeverification00.
    MESSAGE 'Code signature failed one or more validation checks for
        signature mismatch or expiration.' TYPE 'E'.
CATCH /aws1/cx_lmdinvalidcodesigex.
    MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
CATCH /aws1/cx_lmdinvparamvalueex.
    MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
CATCH /aws1/cx_lmdresourceconflictex.
    MESSAGE 'Resource already exists or another operation is in progress.'
        TYPE 'E'.
CATCH /aws1/cx_lmdresourcetonffoundex.
    MESSAGE 'The requested resource does not exist.' TYPE 'E'.
CATCH /aws1/cx_lmdserviceexception.
    MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
        TYPE 'E'.
CATCH /aws1/cx_lmdtoomanyrequestsex.
    MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
ENDTRY.
For API details, see `UpdateFunctionConfiguration` in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

**Scenarios for Lambda using AWS SDKs**

The following code examples show you how to implement common scenarios in Lambda with AWS SDKs. These scenarios show you how to accomplish specific tasks by calling multiple functions within Lambda. Each scenario includes a link to GitHub, where you can find instructions on how to set up and run the code.

**Examples**
- Get started creating and invoking Lambda functions using an AWS SDK

**Get started creating and invoking Lambda functions using an AWS SDK**

The following code examples show how to:

- Create an IAM role and Lambda function, then upload handler code.
- Invoke the function with a single parameter and get results.
- Update the function code and configure with an environment variable.
- Invoke the function with new parameters and get results. Display the returned execution log.
- List the functions for your account, then clean up resources.

For more information, see [Create a Lambda function with the console](#).

**.NET**

**AWS SDK for .NET**

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).
Create methods that perform Lambda actions.

```csharp
namespace LambdaActions;

using Amazon.Lambda;
using Amazon.Lambda.Model;

/// <summary>
/// A class that implements AWS Lambda methods.
/// </summary>
public class LambdaWrapper
{
    private readonly IAmazonLambda _lambdaService;

    /// <summary>
    /// Constructor for the LambdaWrapper class.
    /// </summary>
    /// <param name="lambdaService">An initialized Lambda service client.</param>
    public LambdaWrapper(IAmazonLambda lambdaService)
    {
        _lambdaService = lambdaService;
    }

    /// <summary>
    /// Creates a new Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function.</param>
    /// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3) bucket where the zip file containing the code is located.</param>
    /// <param name="s3Key">The Amazon S3 key of the zip file.</param>
    /// <param name="role">The Amazon Resource Name (ARN) of a role with the appropriate Lambda permissions.</param>
    /// <param name="handler">The name of the handler function.</param>
    /// <returns>The Amazon Resource Name (ARN) of the newly created Lambda function.</returns>
    public async Task<string> CreateLambdaFunctionAsync(
        string functionName,
        string s3Bucket,
        string s3Key,
        string role,
        string handler)
    {
        // Defines the location for the function code.
        // S3Bucket - The S3 bucket where the file containing
```csharp
// the source code is stored.
// S3Key    - The name of the file containing the code.
var functionCode = new FunctionCode
{
    S3Bucket = s3Bucket,
    S3Key = s3Key,
};

var createFunctionRequest = new CreateFunctionRequest
{
    FunctionName = functionName,
    Description = "Created by the Lambda .NET API",
    Code = functionCode,
    Handler = handler,
    Runtime = Runtime.Dotnet6,
    Role = role,
};

var reponse = await _lambdaService.CreateFunctionAsync(createFunctionRequest);
return reponse.FunctionArn;
}

/// <summary>
/// Delete an AWS Lambda function.
/// </summary>
/// <param name="functionName">The name of the Lambda function to delete.</param>
/// <returns>A Boolean value that indicates the success of the action.</returns>
public async Task<bool> DeleteFunctionAsync(string functionName)
{
    var request = new DeleteFunctionRequest
    {
        FunctionName = functionName,
    }; 

    var response = await _lambdaService.DeleteFunctionAsync(request);
    // A return value of NoContent means that the request was processed.
    // In this case, the function was deleted, and the return value is intentionally blank.
}
```
Get started with functions
string returnValue =
    System.Text.Encoding.UTF8.GetString(stream.ToArray());
    return returnValue;
}

/// <summary>
/// Get a list of Lambda functions.
/// </summary>
/// <returns>A list of FunctionConfiguration objects.</returns>
public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
{
    var functionList = new List<FunctionConfiguration>();

    var functionPaginator =
        _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
    await foreach (var function in functionPaginator.Functions)
    {
        functionList.Add(function);
    }

    return functionList;
}

/// <summary>
/// Update an existing Lambda function.
/// </summary>
/// <param name="functionName">The name of the Lambda function to update.</param>
/// <param name="bucketName">The bucket where the zip file containing
/// the Lambda function code is stored.</param>
/// <param name="key">The key name of the source code file.</param>
/// <returns>Async Task.</returns>
public async Task UpdateFunctionCodeAsync(
    string functionName,
    string bucketName,
    string key)
{
    var functionCodeRequest = new UpdateFunctionCodeRequest
    {
        FunctionName = functionName,
        Publish = true,
        S3Bucket = bucketName,
S3Key = key,
};

var response = await _lambdaService.UpdateFunctionConfigurationAsync(request);
Console.WriteLine("The Function was last modified at 
{response.LastModified}.");
}

/// <summary>
/// Update the code of a Lambda function.
/// </summary>
/// <param name="functionName">The name of the function to update.</param>
/// <param name="functionHandler">The code that performs the function's actions.</param>
/// <param name="environmentVariables">A dictionary of environment variables.</param>
/// <returns>A Boolean value indicating the success of the action.</returns>
public async Task<bool> UpdateFunctionConfigurationAsync(
    string functionName,
    string functionHandler,
    Dictionary<string, string> environmentVariables)
{
    var request = new UpdateFunctionConfigurationRequest
    {
        Handler = functionHandler,
        FunctionName = functionName,
        Environment = new Amazon.Lambda.Model.Environment
        { Variables = environmentVariables },
    };

    var response = await _lambdaService.UpdateFunctionConfigurationAsync(request);
    Console.WriteLine(response.LastModified);
}
Create a function that runs the scenario.

global using System.Threading.Tasks;
global using Amazon.IdentityManagement;
global using Amazon.Lambda;
global using LambdaActions;
global using LambdaScenarioCommon;
global using Microsoft.Extensions.DependencyInjection;
global using Microsoft.Extensions.Hosting;
global using Microsoft.Extensions.Logging;
global using Microsoft.Extensions.Logging.Console;
global using Microsoft.Extensions.Logging.Debug;
using Amazon.Lambda.Model;
using Microsoft.Extensions.Configuration;

namespace LambdaBasics;

public class LambdaBasics
{
    private static ILogger logger = null!;

    static async Task Main(string[] args)
    {
        // Set up dependency injection for the Amazon service.
        using var host = Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
                logging.AddFilter("System", LogLevel.Debug)
                .AddFilter<DebugLoggerProvider>("Microsoft",
                    LogLevel.Information)
                .AddFilter<ConsoleLoggerProvider>("Microsoft",
                    LogLevel.Trace))
            .ConfigureServices((_, services) =>
                services.AddAWSService<IAmazonLambda>()
                .AddAWSService<IAmazonIdentityManagementService>()
                .AddTransient<LambdaWrapper>()
                .AddTransient<LambdaRoleWrapper>()
                .AddTransient<UIWrapper>()
            )
            .Build();
var configuration = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .Build();

logger = LoggerFactory.Create(builder => { builder.AddConsole(); })
    .CreateLogger<LambdaBasics>();

var lambdaWrapper = host.Services.GetRequiredService<LambdaWrapper>();
var lambdaRoleWrapper = host.Services.GetRequiredService<LambdaRoleWrapper>();
var uiWrapper = host.Services.GetRequiredService<UIWrapper>();

string functionName = configuration["FunctionName"];  
string roleName = configuration["RoleName"];  
string policyDocument = "{" +
"   \"Version\": \"2012-10-17\"," +
"   \"Statement\": [ " +
"     {" +
"       \"Effect\": \"Allow\"," +
"       \"Principal\": {" +
"       \"Service\": \"lambda.amazonaws.com\" " +
"       }," +
"       \"Action\": \"sts:AssumeRole\" " +
"     }" +
"   "]" +
"}," +
"};

var incrementHandler = configuration["IncrementHandler"];  
var calculatorHandler = configuration["CalculatorHandler"];  
var bucketName = configuration["BucketName"];  
var incrementKey = configuration["IncrementKey"];  
var calculatorKey = configuration["CalculatorKey"];  
var policyArn = configuration["PolicyArn"];  

uiWrapper.DisplayLambdaBasicsOverview();

// Create the policy to use with the AWS Lambda functions and then attach the
// policy to a new role.
```csharp
var roleArn = await lambdaRoleWrapper.CreateLambdaRoleAsync(roleName, policyDocument);

Console.WriteLine("Waiting for role to become active.");
uiWrapper.WaitABit(15, "Wait until the role is active before trying to use it.");

// Attach the appropriate AWS Identity and Access Management (IAM) role policy to the new role.
var success = await lambdaRoleWrapper.AttachLambdaRolePolicyAsync(policyArn, roleName);
uiWrapper.WaitABit(10, "Allow time for the IAM policy to be attached to the role.");

// Create the Lambda function using a zip file stored in an Amazon Simple Storage Service
// (Amazon S3) bucket.
uiWrapper.DisplayTitle("Create Lambda Function");
Console.WriteLine($"Creating the AWS Lambda function: {functionName}.");
var lambdaArn = await lambdaWrapper.CreateLambdaFunctionAsync(
    functionName,
    bucketName,
    incrementKey,
    roleArn,
    incrementHandler);

Console.WriteLine("Waiting for the new function to be available.");
Console.WriteLine($"The AWS Lambda ARN is {lambdaArn}";

// Get the Lambda function.
Console.WriteLine($"Getting the {functionName} AWS Lambda function.");
FunctionConfiguration config;
do {
    config = await lambdaWrapper.GetFunctionAsync(functionName);
    Console.Write("");
} while (config.State != State.Active);

Console.WriteLine($"\nThe function, {functionName} has been created.");
Console.WriteLine($"The runtime of this Lambda function is {config.Runtime}.");
uiWrapper.PressEnter();
```
// List the Lambda functions.
uiWrapper.DisplayTitle("Listing all Lambda functions.");
var functions = await lambdaWrapper.ListFunctionsAsync();
DisplayFunctionList(functions);

uiWrapper.DisplayTitle("Invoke increment function");
Console.WriteLine("Now that it has been created, invoke the Lambda increment function.");
string? value;
do {
    Console.Write("Enter a value to increment: ");
    value = Console.ReadLine();
}
while (string.IsNullOrEmpty(value));

string functionParameters = "{" + 
    "\"action\": \"increment\", " + 
    "\"x\": \"" + value + \"\" + 
    "}";
var answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
Console.WriteLine($"{value} + 1 = {answer}.");

uiWrapper.DisplayTitle("Update function");
Console.WriteLine("Now update the Lambda function code.");
await lambdaWrapper.UpdateFunctionCodeAsync(functionName, bucketName,
calculatorKey);

do {
    config = await lambdaWrapper.GetFunctionAsync(functionName);
    Console.Write(".");
}
while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

await lambdaWrapper.UpdateFunctionConfigurationAsync(
functionName,
calculatorHandler,
new Dictionary<string, string> { { "LOG_LEVEL", "DEBUG" } });

do {
config = await lambdaWrapper.GetFunctionAsync(functionName);
    Console.Write(".");
}
while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

uiWrapper.DisplayTitle("Call updated function");
Console.WriteLine("Now call the updated function...");

bool done = false;

do
{
    string? opSelected;

    Console.WriteLine("Select the operation to perform:");
    Console.WriteLine("\t1. add");
    Console.WriteLine("\t2. subtract");
    Console.WriteLine("\t3. multiply");
    Console.WriteLine("\t4. divide");
    Console.WriteLine("\t0r enter \"q\" to quit.");
    Console.WriteLine("Enter the number (1, 2, 3, 4, or q) of the operation you want to perform: ");
    do
    {
        Console.Write("Your choice? ");
        opSelected = Console.ReadLine();
    } while (opSelected == string.Empty);

    var operation = (opSelected) switch
    {
        "1" => "add",
        "2" => "subtract",
        "3" => "multiply",
        "4" => "divide",
        "q" => "quit",
        _ => "add",
    };

    if (operation == "quit")
    {
        done = true;
    }
    else
```csharp
{
    // Get two numbers and an action from the user.
    value = string.Empty;
    do
    {
        Console.Write("Enter the first value: ");
        value = Console.ReadLine();
    } while (value == string.Empty);

    string? value2;
    do
    {
        Console.Write("Enter a second value: ");
        value2 = Console.ReadLine();
    } while (value2 == string.Empty);

    functionParameters = "{" +
    ""action": ": ", " +
    ""x": ": ", " +
    ""y": ": ", " +
    "}";

    answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
    functionParameters);
    Console.WriteLine("The answer when we {operation} the two numbers is: 
    ");
}

    uiWrapper.PressEnter();
} while (!done);

// Delete the function created earlier.

uiWrapper.DisplayTitle("Clean up resources");
// Detach the IAM policy from the IAM role.
Console.WriteLine("First detach the IAM policy from the role.");
success = await lambdaRoleWrapper.DetachLambdaRolePolicyAsync(policyArn,
    roleName);
uiWrapper.Wa...

    Console.WriteLine("Delete the AWS Lambda function.");
```
success = await lambdaWrapper.DeleteFunctionAsync(functionName);
if (success)
{
    Console.WriteLine("The {functionName} function was deleted.");
}
else
{
    Console.WriteLine("Could not remove the function {functionName}" );
}

// Now delete the IAM role created for use with the functions
// created by the application.
Console.WriteLine("Now we can delete the role that we created.");
success = await lambdaRoleWrapper.DeleteLambdaRoleAsync(roleName);
if (success)
{
    Console.WriteLine("The role has been successfully removed.");
}
else
{
    Console.WriteLine("Couldn't delete the role.");
}

Console.WriteLine("The Lambda Scenario is now complete.");
uiWrapper.PressEnter();

// Displays a formatted list of existing functions returned by the
// LambdaMethods.ListFunctions.
void DisplayFunctionList(List<FunctionConfiguration> functions)
{
    functions.ForEach(functionConfig =>
    {
        Console.WriteLine("{functionConfig.FunctionName}	{functionConfig.Description}");
    });
}

namespace LambdaActions;

using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;
public class LambdaRoleWrapper
{
    private readonly IAmazonIdentityManagementService _lambdaRoleService;

    public LambdaRoleWrapper(IAmazonIdentityManagementService lambdaRoleService)
    {
        _lambdaRoleService = lambdaRoleService;
    }

    /// <summary>
    /// Attach an AWS Identity and Access Management (IAM) role policy to the
    /// IAM role to be assumed by the AWS Lambda functions created for the
    /// scenario.
    /// </summary>
    /// <param name="policyArn">The Amazon Resource Name (ARN) of the IAM
    /// policy.</param>
    /// <param name="roleName">The name of the IAM role to attach the IAM policy
to.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> AttachLambdaRolePolicyAsync(string policyArn, string
roleName)
    {
        var response = await _lambdaRoleService.AttachRolePolicyAsync(new
AttachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
    }

    /// <summary>
    /// Create a new IAM role.
    /// </summary>
    /// <param name="roleName">The name of the IAM role to create.</param>
    /// <param name="policyDocument">The policy document for the new IAM role.</param>
    /// <returns>A string representing the ARN for newly created role.</returns>
    public async Task<string> CreateLambdaRoleAsync(string roleName, string
policyDocument)
    {
        var request = new CreateRoleRequest
        {
            AssumeRolePolicyDocument = policyDocument,
            RoleName = roleName,
        };
    }
var response = await _lambdaRoleService.CreateRoleAsync(request);
return response.Role.Arn;
}

/// <summary>
/// Deletes an IAM role.
/// </summary>
/// <param name="roleName">The name of the role to delete.</param>
/// <returns>A Boolean value indicating the success of the operation.</returns>
public async Task<bool> DeleteLambdaRoleAsync(string roleName)
{
var request = new DeleteRoleRequest
{
    RoleName = roleName,
};

var response = await _lambdaRoleService.DeleteRoleAsync(request);
}

public async Task<bool> DetachLambdaRolePolicyAsync(string policyArn, string
roleName)
{
    var response = await _lambdaRoleService.DetachRolePolicyAsync(new
DetachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
}

}
Console.WriteLine("This example application does the following:");
Console.WriteLine("\t1. Creates an AWS Identity and Access Management (IAM) role that will be assumed by the functions we create.");
Console.WriteLine("\t2. Attaches an IAM role policy that has Lambda permissions.");
Console.WriteLine("\t3. Creates a Lambda function that increments the value passed to it.");
Console.WriteLine("\t4. Calls the increment function and passes a value.");
Console.WriteLine("\t5. Updates the code so that the function is a simple calculator.");
Console.WriteLine("\t6. Calls the calculator function with the values entered.");
Console.WriteLine("\t7. Deletes the Lambda function.");
Console.WriteLine("\t7. Detaches the IAM role policy.");
Console.WriteLine("\t8. Deletes the IAM role.");
PressEnter();

/// <summary>
/// Display a message and wait until the user presses enter.
/// </summary>
public void PressEnter()
{
    Console.Write("\nPress <Enter> to continue. ");
    _ = Console.ReadLine();
    Console.WriteLine();
}

/// <summary>
/// Pad a string with spaces to center it on the console display.
/// </summary>
/// <param name="strToCenter">The string to be centered.</param>
/// <returns>The padded string.</returns>
public string CenterString(string strToCenter)
{
    var padAmount = (Console.WindowWidth - strToCenter.Length) / 2;
    var leftPad = new string(' ', padAmount);
    return $"{leftPad}{strToCenter}";
}

/// <summary>
/// Display a line of hyphens, the centered text of the title and another line of hyphens.
/// </summary>
Define a Lambda handler that increments a number.

```csharp
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
[assembly:
    LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer))]

namespace LambdaIncrement;

public class Function
{
    // Get started with functions
```
/// <summary>
/// A simple function increments the integer parameter.
/// </summary>
/// <param name="input">A JSON string containing an action, which must be "increment" and a string representing the value to increment.</param>
/// <param name="context">The context object passed by Lambda containing information about invocation, function, and execution environment.</param>
/// <returns>A string representing the incremented value of the parameter.</returns>
public int FunctionHandler(Dictionary<string, string> input, ILambdaContext context)
{
    if (input["action"] == "increment")
    {
        int inputValue = Convert.ToInt32(input["x"]);
        return inputValue + 1;
    }
    else
    {
        return 0;
    }
}

Define a second Lambda handler that performs arithmetic operations.

using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
[assembly:
]

namespace LambdaCalculator;

public class Function
{
    /// <summary>
    /// Define a second Lambda handler that performs arithmetic operations.
    /// </summary>
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext context)
    {
        if (input["action"] == "increment")
        {
            int inputValue = Convert.ToInt32(input["x"]);
            return inputValue + 1;
        }
        else
        {
            return 0;
        }
    }
}
/// A simple function that takes two number in string format and performs
/// the requested arithmetic function.
/// <summary>
/// <param name="input">JSON data containing an action, and x and y values.
/// Valid actions include: add, subtract, multiply, and divide.</param>
/// <param name="context">The context object passed by Lambda containing
/// information about invocation, function, and execution environment.</param>
/// <returns>A string representing the results of the calculation.</returns>
/// </summary>
public int FunctionHandler(Dictionary<string, string> input, ILambdaContext context)
{
    var action = input["action"];
    int x = Convert.ToInt32(input["x"]);
    int y = Convert.ToInt32(input["y"]);
    int result;
    switch (action)
    {
    case "add":
        result = x + y;
        break;
    case "subtract":
        result = x - y;
        break;
    case "multiply":
        result = x * y;
        break;
    case "divide":
        if (y == 0)
        {
            Console.Error.WriteLine("Divide by zero error.");
            result = 0;
        }
        else
        {
            result = x / y;
            break;
        }
    default:
        Console.Error.WriteLine("{action} is not a valid operation.");
        result = 0;
        break;
    }
    return result;
}
For API details, see the following topics in *AWS SDK for .NET API Reference*.

- [CreateFunction](#)
- [DeleteFunction](#)
- [GetFunction](#)
- [Invoke](#)
- [ListFunctions](#)
- [UpdateFunctionCode](#)
- [UpdateFunctionConfiguration](#)

### C++

**SDK for C++**

```cpp
#include "Lambda.h"

//! Get started with functions scenario.
/*!\param clientConfig: AWS client configuration.
\param bool: Successful completion.
*/
bool AwsDoc::Lambda::getStartedWithFunctionsScenario(
    const Aws::Client::ClientConfiguration &clientConfig) {

    Aws::Lambda::LambdaClient client(clientConfig);

    // 1. Create an AWS Identity and Access Management (IAM) role for Lambda function.
    Aws::String roleArn;
    if (!getIamRoleArn(roleArn, clientConfig)) {
        return false;
    }
```

[Note]
There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-cpp).

Get started with functions
// 2. Create a Lambda function.
int seconds = 0;
do {
    Aws::Lambda::Model::CreateFunctionRequest request;
    request.SetFunctionName(LAMBDA_NAME);
    request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
    #if USE_CPP_LAMBDA_FUNCTION
    request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
    request.SetTimeout(15);
    request.SetMemorySize(128);
    #else
    request.SetRuntime(Aws::Lambda::Model::Runtime::python3_8);
    #endif // defined(architecture)
    request.SetRole(roleArn);
    request.SetHandler(LAMBDA_HANDLER_NAME);
    request.SetPublish(true);
    Aws::Lambda::Model::FunctionCode code;
    std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                            std::ios_base::in | std::ios_base::binary);
    if (!ifstream.is_open()) {
        std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." "
    } else {
    #if USE_CPP_LAMBDA_FUNCTION
    std::cerr
        "The cpp Lambda function must be built following the 
        instructions in the cpp_lambda/README.md file. "
        "
        std::endl;
    #endif
    deleteIamRole(clientConfig);
    return false;
}
Aws::StringStream buffer;
buffer << ifstream.rdbuf();

code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
buffer.str().c_str(),

buffer.str().length()));
request.SetCode(code);

Aws::Lambda::Model::CreateFunctionOutcome outcome =
client.CreateFunction(
    request);

if (outcome.IsSuccess()) {
    std::cout << "The lambda function was successfully created. " <<

    seconds
    << " seconds elapsed." << std::endl;
    break;
}
else if (outcome.GetError().GetErrorType() ==
    Aws::Lambda::LambdaErrors::INVALID_PARAMETER_VALUE &&
    outcome.GetError().GetMessage().find("role") >= 0) {
    if ((seconds % 5) == 0) { // Log status every 10 seconds.
        std::cout
            << "Waiting for the IAM role to become available as a
CreateFunction parameter. "
            << seconds
            << " seconds elapsed." << std::endl;

        std::cout << outcome.GetError().GetMessage() << std::endl;
    }
}
else {
    std::cerr << "Error with CreateFunction. "

    << outcome.GetError().GetMessage()
    << std::endl;
    deleteIamRole(clientConfig);
    return false;
}
    ++seconds;
    std::this_thread::sleep_for(std::chrono::seconds(1));
} while (60 > seconds);
std::cout << "The current Lambda function increments 1 by an input."
<< std::endl;

// 3. Invoke the Lambda function.
{
    int increment = askQuestionForInt("Enter an increment integer: ");

    Aws::Lambda::Model::InvokeResult invokeResult;
    Aws::Utils::Json::JsonValue jsonPayload;
    jsonPayload.WithString("action", "increment");
    jsonPayload.WithInteger("number", increment);
    if (invokeLambdaFunction(jsonPayload, Aws::Lambda::Model::LogType::Tail,
                               invokeResult, client)) {
        Aws::Utils::Json::JsonValue jsonValue(invokeResult.GetPayload());
        Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
            jsonValue.View().GetAllObjects();
        auto iter = values.find("result");
        if (iter != values.end() && iter->second.IsIntegerType()) {
            std::cout << INCREMENT_RESULT_PREFIX
                        << iter->second.AsInteger() << std::endl;
        }
    } else {
        std::cout << "There was an error in execution. Here is the log."
                  << std::endl;
        Aws::Utils::ByteBuffer buffer =
            Aws::Utils::HashingUtils::Base64Decode(          invokeResult.GetLogResult());
        std::cout << "With log " << buffer.GetUnderlyingData() << std::endl;
    }
}

std::cout
    << "The Lambda function will now be updated with new code. Press
return to continue, ";
    Aws::String answer;
    std::getline(std::cin, answer);

// 4. Update the Lambda function code.
{
    Aws::Lambda::Model::UpdateFunctionCodeRequest request;
    }
request.SetFunctionName(LAMBDA_NAME);
std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
    std::ios_base::in | std::ios_base::binary);
if (!ifstream.is_open()) {
    std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." << 
    std::endl;

#if USE_CPP_LAMBDA_FUNCTION
    std::cerr
        << "The cpp Lambda function must be built following the
    instructions in the cpp_lambda/README.md file. "
        << std::endl;
#endif
    deleteLambdaFunction(client);
    deleteIamRole(clientConfig);
    return false;
}

Aws::StringStream buffer;
buffer << ifstream.rdbuf();
request.SetZipFile(
    Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
        buffer.str().length()));
request.SetPublish(true);

Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome = 
client.UpdateFunctionCode( 
    request);

if (outcome.IsSuccess()) {
    std::cout << "The lambda code was successfully updated." << 
    std::endl;
} 
else {
    std::cerr << "Error with Lambda::UpdateFunctionCode. "
        << outcome.GetError().GetMessage()
        << std::endl;
}

std::cout
    << "This function uses an environment variable to control the logging
level."
    << std::endl;
UpdateFunctionConfiguration will be used to set the LOG_LEVEL to DEBUG.

seconds = 0;

// 5. Update the Lambda function configuration.
do {
  ++seconds;
  std::this_thread::sleep_for(std::chrono::seconds(1));
  Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
  request.SetFunctionName(LAMBDA_NAME);
  Aws::Lambda::Model::Environment environment;
  environment.AddVariables("LOG_LEVEL", "DEBUG");
  request.SetEnvironment(environment);

  Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
  client.UpdateFunctionConfiguration(request);

  if (outcome.IsSuccess()) {
    std::cout << "The lambda configuration was successfully updated."
      << std::endl;
    break;
  }

  // RESOURCE_IN_USE: function code update not completed.
  else if (outcome.GetError().GetErrorType() !=
    Aws::Lambda::LambdaErrors::RESOURCE_IN_USE) {
    if ((seconds % 10) == 0) { // Log status every 10 seconds.
      std::cout << "Lambda function update in progress. After " <<
        seconds
      " seconds elapsed." << std::endl;
    }
  }
  else {
    std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
      << outcome.GetError().GetMessage()
    << std::endl;
  }
} while (0 < seconds);

if (0 > seconds) {
std::cerr << "Function failed to become active." << std::endl;
} else {
    std::cout << "Updated function active after " << seconds << " seconds."
    << std::endl;
}

std::cout
    << "\nThe new code applies an arithmetic operator to two variables, x
an y."
    << std::endl;
std::vector<Aws::String> operators = {"plus", "minus", "times", "divided-by"};
for (size_t i = 0; i < operators.size(); ++i) {
    std::cout << "   " << i + 1 << " " << operators[i] << std::endl;
}

// 6. Invoke the updated Lambda function.
do {
    int operatorIndex = askQuestionForIntRange("Select an operator index 1 - 4 ", 1,
        4);
    int x = askQuestionForInt("Enter an integer for the x value ");
    int y = askQuestionForInt("Enter an integer for the y value ");

    Aws::Utils::Json::JsonValue calculateJsonPayload;
    calculateJsonPayload.WithString("action", operators[operatorIndex - 1]);
    calculateJsonPayload.WithInteger("x", x);
    calculateJsonPayload.WithInteger("y", y);
    Aws::Lambda::Model::InvokeResult calculatedResult;
    if (invokeLambdaFunction(calculateJsonPayload,
        Aws::Lambda::Model::LogType::Tail,
        calculatedResult, client)) {
        Aws::Utils::Json::JsonValue jsonValue(calculatedResult.GetPayload());
        Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
            jsonValue.View().GetAllObjects();
        auto iter = values.find("result");
        if (iter != values.end() && iter->second.IsIntegerType()) {
            std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                << operators[operatorIndex - 1] << " "
                << y << " is " << iter->second.AsInteger() <<
            std::endl;
        }
    }
else if (iter != values.end() && iter->second.IsFloatingPointType())
{
    std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
               << operators[operatorIndex - 1] << " "
               << y << " is " << iter->second.AsDouble() << std::endl;
}
else {
    std::cout << "There was an error in execution. Here is the log."
              << std::endl;
    Aws::Utils::ByteBuffer buffer =
        Aws::Utils::HashingUtils::Base64Decode(
            calculatedResult.GetLogResult());
    std::cout << "With log " << buffer.GetUnderlyingData() << std::endl;
}

answer = askQuestion("Would you like to try another operation? (y/n) ");
} while (answer == "y");

std::cout
    << "A list of the lambda functions will be retrieved. Press return to continue, ";
std::getline(std::cin, answer);

// 7. List the Lambda functions.

std::vector<Aws::String> functions;
Aws::String marker;

do {
    Aws::Lambda::Model::ListFunctionsRequest request;
    if (!marker.empty()) {
        request.SetMarker(marker);
    }

    Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
        request);

    if (outcome.IsSuccess()) {
        const Aws::Lambda::Model::ListFunctionsResult &result =
            outcome.GetResult();
        std::cout << result.GetFunctions().size() << " lambda functions were retrieved." << std::endl;
    }
for (const Aws::Lambda::Model::FunctionConfiguration &functionConfiguration: result.GetFunctions()) {
    functions.push_back(functionConfiguration.GetFunctionName());
    std::cout << functions.size() << " " << functionConfiguration.GetDescription() << std::endl;
    std::cout << " " <<
    Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
        functionConfiguration.GetRuntime()) << ": "
    << functionConfiguration.GetHandler() << std::endl;
}

marker = result.GetNextMarker();
}

else {
    std::cerr << "Error with Lambda::ListFunctions. " << outcome.GetError().GetMessage() << std::endl;
}
} while (!marker.empty());

if (!functions.empty()) {
    std::stringstream question;
    question << "Choose a function to retrieve between 1 and " << functions.size() << " ";
    int functionIndex = askQuestionForIntRange(question.str(), 1,
    static_cast<int>(functions.size()));

    Aws::String functionName = functions[functionIndex - 1];

    Aws::Lambda::Model::GetFunctionRequest request;
    request.SetFunctionName(functionName);

    Aws::Lambda::Model::GetFunctionOutcome outcome = client.GetFunction(request);

    if (outcome.IsSuccess()) {
        std::cout << "Function retrieve.\n" << outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()" <<
    } else {
        std::cerr << "Error with Lambda::ListFunctions. " << outcome.GetError().GetMessage() << std::endl;
    }
} while (!marker.empty());
<< std::endl;
}
}

std::cout << "The resources will be deleted. Press return to continue, ";
std::getline(std::cin, answer);

// 9. Delete the Lambda function.
bool result = deleteLambdaFunction(client);

// 10. Delete the IAM role.
return result && deleteIamRole(clientConfig);
}

/*!
Routine which invokes a Lambda function and returns the result.

\param jsonPayload: Payload for invoke function.
\param logType: Log type setting for invoke function.
\param invokeResult: InvokeResult object to receive the result.
\param client: Lambda client.
\return bool: Successful completion.
*/
bool
AwsDoc::Lambda::invokeLambdaFunction(const Aws::Utils::Json::JsonValue
  &jsonPayload, const Aws::Lambda::LambdaClient &client) {
  int seconds = 0;
  bool result = false;
  /*
  * In this example, the Invoke function can be called before recently created
  * resources are
  * available. The Invoke function is called repeatedly until the resources
  * are
  * available.
  */
  do {
    seconds++;
    // Call the function
    Aws::Lambda::InvokeResult invokeResult;
    // Call the function
    result = true;
  } while (result && seconds < MAXIMUM_ATTEMPTS);
  return result && seconds < MAXIMUM_ATTEMPTS;
}
```cpp
Aws::Lambda::Model::InvokeRequest request;
request.SetFunctionName(LAMBDA_NAME);
request.SetLogType(logType);
std::shared_ptr<Aws::IOStream> payload =
  Aws::MakeShared<Aws::StringStream>(
    "FunctionTest");
*payload << jsonPayload.View().WriteReadable();
request.SetBody(payload);
request.SetContentType("application/json");
Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

if (outcome.IsSuccess()) {
    invokeResult = std::move(outcome.GetResult());
    result = true;
    break;
}

// ACCESS_DENIED: because the role is not available yet.
// RESOURCE_CONFLICT: because the Lambda function is being created or
updated.
else if ((outcome.GetError().GetErrorType() ==
    Aws::Lambda::LambdaErrors::ACCESS_DENIED) ||
    (outcome.GetError().GetErrorType() ==
    Aws::Lambda::LambdaErrors::RESOURCE_CONFLICT)) {
    if ((seconds % 5) == 0) { // Log status every 10 seconds.
        std::cout << "Waiting for the invoke api to be available, status
" <<
            ((outcome.GetError().GetErrorType() ==
                Aws::Lambda::LambdaErrors::ACCESS_DENIED ?
                "ACCESS_DENIED" : "RESOURCE_CONFLICT")) << ". " <<
        seconds
            << " seconds elapsed." << std::endl;
    }
}
else {
    std::cerr << "Error with Lambda::InvokeRequest. "
        << outcome.GetError().GetMessage()
        << std::endl;
    break;
}
++seconds;
std::this_thread::sleep_for(std::chrono::seconds(1));
} while (seconds < 60);
```
For API details, see the following topics in *AWS SDK for C++ API Reference*.

- CreateFunction
- DeleteFunction
- GetFunction
- Invoke
- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

Go

SDK for Go V2

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://aws.amazon.com/code-examples/).

Create an interactive scenario that shows you how to get started with Lambda functions.

```go
// GetStartedFunctionsScenario shows you how to use AWS Lambda to perform the following
// actions:
// 1. Create an AWS Identity and Access Management (IAM) role and Lambda function, then upload handler code.
// 2. Invoke the function with a single parameter and get results.
// 3. Update the function code and configure with an environment variable.
// 4. Invoke the function with new parameters and get results. Display the returned execution log.
// 5. List the functions for your account, then clean up resources.

type GetStartedFunctionsScenario struct {
    sdkConfig     aws.Config
    // ...
functionWrapper actions.FunctionWrapper
questioner demotools.IQuestioner
helper IScenarioHelper
isTestRun bool
}

// NewGetStartedFunctionsScenario constructs a GetStartedFunctionsScenario instance from a configuration.
// It uses the specified config to get a Lambda client and create wrappers for the actions used in the scenario.
func NewGetStartedFunctionsScenario(sdkConfig aws.Config, questioner demotools.IQuestioner, helper IScenarioHelper) GetStartedFunctionsScenario {
lambdaClient := lambda.NewFromConfig(sdkConfig)
return GetStartedFunctionsScenario{
sdkConfig: sdkConfig,
functionWrapper: actions.FunctionWrapper{LambdaClient: lambdaClient},
questioner: questioner,
helper: helper,
}
}

// Run runs the interactive scenario.
func (scenario GetStartedFunctionsScenario) Run() {
defer func() {
    if r := recover(); r != nil {
        log.Printf("Something went wrong with the demo.\n")
    }
}()

log.Println(strings.Repeat("-", 88))
log.Println("Welcome to the AWS Lambda get started with functions demo.")
log.Println(strings.Repeat("-", 88))

role := scenario.GetOrCreateRole()
funcName := scenario.CreateFunction(role)
scenario.InvokeIncrement(funcName)
scenario.UpdateFunction(funcName)
scenario.InvokeCalculator(funcName)
scenario.ListFunctions()
scenario.Cleanup(role, funcName)

log.Println(strings.Repeat("-", 88))
log.Println("Thanks for watching!")
log.Println(strings.Repeat("-", 88))
}

// GetOrCreateRole checks whether the specified role exists and returns it if it does.
// Otherwise, a role is created that specifies Lambda as a trusted principal.
// The AWSLambdaBasicExecutionRole managed policy is attached to the role and the role
// is returned.
func (scenario GetStartedFunctionsScenario) GetOrCreateRole() *iamtypes.Role {
    var role *iamtypes.Role
    iamClient := iam.NewFromConfig(scenario.sdkConfig)
    log.Println("First, we need an IAM role that Lambda can assume.
    roleName := scenario.questioner.Ask("Enter a name for the role:",
    demotools.NotEmpty{})
    getOutput, err := iamClient.GetRole(context.TODO(), &iam.GetRoleInput{
        RoleName: aws.String(roleName))
    if err != nil {
        var noSuch *iamtypes.NoSuchEntityException
        if errors.As(err, &noSuch) {
            log.Printf("Role %v doesn't exist. Creating it....\n", roleName)
        } else {
            log.Panicf("Couldn't check whether role %v exists. Here's why: %v\n",
                roleName, err)
        }
    } else {
        role = getOutput.Role
        log.Printf("Found role %v.\n", *role.RoleName)
    }
    if role == nil {
        trustPolicy := PolicyDocument{
            Version: "2012-10-17",
            Statement: []PolicyStatement{{
                Effect: "Allow",
                Principal: map[string]string{"Service": "lambda.amazonaws.com"},
                Action: []string{"sts:AssumeRole"},
            }},
        }
        policyArn := "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
        createOutput, err := iamClient.CreateRole(context.TODO(), &iam.CreateRoleInput{
            AssumeRolePolicyDocument: aws.String(trustPolicy.String()),
            RoleName: aws.String(roleName),
        })
    }
if err != nil {
    log.Panicf("Couldn't create role %v. Here's why: %v\n", roleName, err)
}
role = createOutput.Role
_, err = iamClient.AttachRolePolicy(context.TODO(), &iam.AttachRolePolicyInput{
    PolicyArn: aws.String(policyArn),
    RoleName:  aws.String(roleName),
})
if err != nil {
    log.Panicf("Couldn't attach a policy to role %v. Here's why: %v\n", roleName, err)
}
log.Printf("Created role %v.\n", *role.RoleName)
log.Println("Let's give AWS a few seconds to propagate resources...")
scenario.helper.Pause(10)
}
log.Println(strings.Repeat("-", 88))
return role
}

// CreateFunction creates a Lambda function and uploads a handler written in
// Python.
// The code for the Python handler is packaged as a []byte in .zip format.
func (scenario GetStartedFunctionsScenario) CreateFunction(role *iamtypes.Role) string {
    log.Println("Let's create a function that increments a number.\n" +
        "The function uses the 'lambda_handler_basic.py' script found in the \n" +
        "'handlers' directory of this project."\n    )
    funcName := scenario.questioner.Ask("Enter a name for the Lambda function:",
        demotools.NotEmpty{})
    zipPackage := scenario.helper.CreateDeploymentPackage("lambda_handler_basic.py",
        fmt.Sprintf("%v.py", funcName))
    log.Printf("Creating function %v and waiting for it to be ready.", funcName)
    funcState := scenario.functionWrapper.CreateFunction(funcName,
        fmt.Sprintf("%v.lambda_handler", funcName),
        role.Arn, zipPackage)
    log.Printf("Your function is %v.", funcState)
    log.Println(strings.Repeat("-", 88))
    return funcName
}

// InvokeIncrement invokes a Lambda function that increments a number. The function
// parameters are contained in a Go struct that is used to serialize the parameters to
// a JSON payload that is passed to the function.
// The result payload is deserialized into a Go struct that contains an int
// value.
func (scenario GetStartedFunctionsScenario) InvokeIncrement(funcName string) {
    parameters := actions.IncrementParameters{Action: "increment"}
    log.Println("Let's invoke our function. This function increments a number.")
    parameters.Number = scenario.questioner.AskInt("Enter a number to increment:",
                                                   demotools.NotEmpty{})
    log.Printf("Invoking %v with %v...
", funcName, parameters.Number)
    invokeOutput := scenario.functionWrapper.Invoke(funcName, parameters, false)
    var payload actions.LambdaResultInt
    err := json.Unmarshal(invokeOutput.Payload, &payload)
    if err != nil {
        log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
                   funcName, err)
    }
    log.Printf("Invoking %v with %v returned %v.
", funcName, parameters.Number,
             payload)
    log.Println(strings.Repeat("-", 88))
}

// UpdateFunction updates the code for a Lambda function by uploading a simple
// arithmetic calculator written in Python. The code for the Python handler is packaged as a
// []byte in .zip format.
// After the code is updated, the configuration is also updated with a new log
// level that instructs the handler to log additional information.
func (scenario GetStartedFunctionsScenario) UpdateFunction(funcName string) {
    log.Println("Let's update the function to an arithmetic calculator.\n" +
               "The function uses the 'lambda_handler_calculator.py' script found in the\n" +
               "'handlers' directory of this project."")
    scenario.questioner.Ask("Press Enter when you're ready.
")
    log.Println("Creating deployment package...
")
    zipPackage :=
    scenario.helper.CreateDeploymentPackage("lambda_handler_calculator.py",
                                             fmt.Sprintf("%v.py", funcName))
    log.Println("...and updating the Lambda function and waiting for it to be
ready.
")
    funcState := scenario.functionWrapper.UpdateFunctionCode(funcName, zipPackage)
    log.Printf("Updated function %v. Its current state is %v.\n", funcName, funcState)
    log.Println("This function uses an environment variable to control logging
level."")
}
log.Println("Let's set it to DEBUG to get the most logging.")
scenario.functionWrapper.UpdateFunctionConfiguration(funcName, map[string]string{"LOG_LEVEL": "DEBUG"})
log.Println(strings.Repeat("-", 88))
}

// InvokeCalculator invokes the Lambda calculator function. The parameters are stored in a
// Go struct that is used to serialize the parameters to a JSON payload. That payload is then passed
// to the function.
// The result payload is deserialized to a Go struct that stores the result as either an
// int or float32, depending on the kind of operation that was specified.
func (scenario *GetStartedFunctionsScenario) InvokeCalculator(funcName string) {
    wantInvoke := true
    choices := []string{"plus", "minus", "times", "divided-by"}
    for wantInvoke {
        choice := scenario.questioner.AskChoice("Select an arithmetic operation:\n", choices)
        x := scenario.questioner.AskInt("Enter a value for x:\n", demotools.NotEmpty{})
        y := scenario.questioner.AskInt("Enter a value for y:\n", demotools.NotEmpty{})
        log.Printf("Invoking %v %v %v...", x, choices[choice], y)
        calcParameters := actions.CalculatorParameters{
            Action: choices[choice],
            X:      x,
            Y:      y,
        }
        invokeOutput := scenario.functionWrapper.Invoke(funcName, calcParameters, true)
        var any
        if choice == 3 { // divide-by results in a float.
            any = actions.LambdaResultFloat{}
        } else {
            any = actions.LambdaResultInt{}
        }
        err := json.Unmarshal(invokeOutput.Payload, &any)
        if err != nil {
            log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n", funcName, err)
        }
        log.Printf("Invoking %v with %v %v %v returned %v.%v\n", funcName,
            calcParameters.X, calcParameters.Action, calcParameters.Y, payload)
        scenario.questioner.Ask("Press Enter to see the logs from the call.")
if err != nil {
    log.Panicf("Couldn't decode log result. Here's why: %v\n", err)
}
log.Println(string(logRes))
wantInvoke = scenario.questioner.AskBool("Do you want to calculate again? (y/n)", "y")
log.Println(strings.Repeat("-", 88))

// ListFunctions lists up to the specified number of functions for your account.
func (scenario GetStartedFunctionsScenario) ListFunctions() {
    count := scenario.questioner.AskInt("Let's list functions for your account. How many do you want to see?", demotools.NotEmpty{)}
    functions := scenario.functionWrapper.ListFunctions(count)
    log.Printf("Found %v functions:", len(functions))
    for _, function := range functions {
        log.Printf("\t%v", *function.FunctionName)
    }
    log.Println(strings.Repeat("-", 88))
}

// Cleanup removes the IAM and Lambda resources created by the example.
func (scenario GetStartedFunctionsScenario) Cleanup(role *iamtypes.Role, funcName string) {
    if scenario.questioner.AskBool("Do you want to clean up resources created for this example? (y/n)", "y") {
        iamClient := iam.NewFromConfig(scenario.sdkConfig)
        policiesOutput, err := iamClient.ListAttachedRolePolicies(context.TODO(), &iam.ListAttachedRolePoliciesInput{RoleName: role.RoleName})
        if err != nil {
            log.Panicf("Couldn't get policies attached to role %v. Here's why: %v\n", *role.RoleName, err)
        }
        for _, policy := range policiesOutput.AttachedPolicies {
            _, err = iamClient.DetachRolePolicy(context.TODO(), &iam.DetachRolePolicyInput{
                PolicyArn: policy.PolicyArn, RoleName: role.RoleName,
            })
            if err != nil {
                log.Panicf("Couldn't detach policy %v from role %v. Here's why: %v\n", *policy.PolicyArn, *role.RoleName, err)
Create a struct that wraps individual Lambda actions.

```go
// FunctionWrapper encapsulates function actions used in the examples. // It contains an AWS Lambda service client that is used to perform user actions. type FunctionWrapper struct {
    LambdaClient *lambda.Client
}

// GetFunction gets data about the Lambda function specified by functionName. func (wrapper FunctionWrapper) GetFunction(functionName string) types.State { var state types.State
    funcOutput, err := wrapper.LambdaClient.GetFunction(context.TODO(), &lambda.GetFunctionInput{
        FunctionName: aws.String(functionName),
    })
    if err != nil {
        log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
    } else {
        state = funcOutput.Configuration.State
    }
    return state
}
```
// CreateFunction creates a new Lambda function from code contained in the
// zipPackage. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
// until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(functionName string, handlerName
string,
iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
var state types.State
_, err := wrapper.LambdaClient.CreateFunction(context.TODO(),
&lambda.CreateFunctionInput{
    FunctionName: aws.String(functionName),
    Role:         iamRoleArn,
    Handler:      aws.String(handlerName),
    Publish:      true,
    Runtime:      types.RuntimePython38,
})
if err != nil {
    var resConflict *types.ResourceConflictException
    if errors.As(err, &resConflict) {
        log.Printf("Function %v already exists.\n", functionName)
        state = types.StateActive
    } else {
        log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
    }
} else {
    waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
    funcOutput, err := waiter.WaitForOutput(context.TODO(),
&lambda.GetFunctionInput{
    FunctionName: aws.String(functionName)}, 1*time.Minute)
    if err != nil {
        log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
functionName, err)
    } else {
        state = funcOutput.Configuration.State
    }
}
return state
}

// UpdateFunctionCode updates the code for the Lambda function specified by
// functionName.
// The existing code for the Lambda function is entirely replaced by the code in
// the
// zipPackage buffer. After the update action is called, a
// lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(functionName string, zipPackage
*bytes.Buffer) types.State {
    var state types.State
    _, err := wrapper.LambdaClient.UpdateFunctionCode(context.TODO(),
&lambda.UpdateFunctionCodeInput{
    FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
})
    if err != nil {
        log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
functionName, err)
    } else {
        waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
        funcOutput, err := waiter.WaitForOutput(context.TODO(),
&lambda.GetFunctionInput{
    FunctionName: aws.String(functionName)}, 1*time.Minute)
        if err != nil {
            log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
functionName, err)
        } else {
            state = funcOutput.Configuration.State
        }
    }
    return state
}

// UpdateFunctionConfiguration updates a map of environment variables configured
for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(functionName string, envVars map[string]string) {
    _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(context.TODO(), &lambda.UpdateFunctionConfigurationInput{
        FunctionName: aws.String(functionName),
        Environment: &types.Environment{Variables: envVars},
    })
    if err != nil {
        log.Panicf("Couldn't update configuration for %v. Here's why: %v", functionName, err)
    }
}

// ListFunctions lists up to maxItems functions for the account. This function uses a
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(maxItems int) []types.FunctionConfiguration {
    var functions []types.FunctionConfiguration
    paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient, &lambda.ListFunctionsInput{
        MaxItems: aws.Int32(int32(maxItems)),
    })
    for paginator.HasMorePages() && len(functions) < maxItems {
        pageOutput, err := paginator.NextPage(context.TODO())
        if err != nil {
            log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
        }
        functions = append(functions, pageOutput.Functions...)
    }
    return functions
}

// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(functionName string) {
        FunctionName: aws.String(functionName),
    })
    if err != nil {

// Invoke invokes the Lambda function specified by functionName, passing the
// parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
tells
// Lambda to include the last few log lines in the returned result.
func (wrapper FunctionWrapper) Invoke(functionName string, parameters any, getLog bool) *lambda.InvokeOutput {
    logType := types.LogTypeNone
    if getLog {
        logType = types.LogTypeTail
    }
    payload, err := json.Marshal(parameters)
    if err != nil {
        log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
    }
    invokeOutput, err := wrapper.LambdaClient.Invoke(context.TODO(),
        &lambda.InvokeInput{
            FunctionName: aws.String(functionName),
            LogType:      logType,
            Payload:      payload,
        })
    if err != nil {
        log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
    }
    return invokeOutput
}

// IncrementParameters is used to serialize parameters to the increment Lambda
// handler.
type IncrementParameters struct {
    Action string `json:"action"
    Number int    `json:"number"
}

// CalculatorParameters is used to serialize parameters to the calculator Lambda
// handler.
Create a struct that implements functions to help run the scenario.

// IScenarioHelper abstracts I/O and wait functions from a scenario so that they can be mocked for unit testing.
type IScenarioHelper interface {
    Pause(secs int)
    CreateDeploymentPackage(sourceFile string, destinationFile string) *bytes.Buffer
}

// ScenarioHelper lets the caller specify the path to Lambda handler functions.
type ScenarioHelper struct {
    HandlerPath string
}

// Pause waits for the specified number of seconds.
func (helper *ScenarioHelper) Pause(secs int) {
    time.Sleep(time.Duration(secs) * time.Second)
}

// CreateDeploymentPackage creates an AWS Lambda deployment package from a source file. The deployment package is stored in .zip format in a bytes.Buffer. The buffer can be
// used to pass a []byte to Lambda when creating the function.
// The specified destinationFile is the name to give the file when it's deployed
to Lambda.
func (helper *ScenarioHelper) CreateDeploymentPackage(sourceFile string,
destinationFile string) *bytes.Buffer {
    var err error
    buffer := &bytes.Buffer{}
    writer := zip.NewWriter(buffer)
    zFile, err := writer.Create(destinationFile)
    if err != nil {
        log.Panicf("Couldn't create destination archive %v. Here's why: %v\n",
                    destinationFile, err)
    }
    sourceBody, err := os.ReadFile(fmt.Sprintf("%v/%v", helper.HandlerPath,
sourceFile))
    if err != nil {
        log.Panicf("Couldn't read handler source file %v. Here's why: %v\n",
                    sourceFile, err)
    } else {
        _, err = zFile.Write(sourceBody)
        if err != nil {
            log.Panicf("Couldn't write handler %v to zip archive. Here's why: %v\n",
                        sourceFile, err)
        }
    }
    err = writer.Close()
    if err != nil {
        log.Panicf("Couldn't close zip writer. Here's why: %v\n", err)
    }
    return buffer
}

Define a Lambda handler that increments a number.

import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
Define a second Lambda handler that performs arithmetic operations.

```python
import logging
import os

logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda function.
ACTIONS = {
    'plus': lambda x, y: x + y,
    'minus': lambda x, y: x - y,
    'times': lambda x, y: x * y,
    'divided-by': lambda x, y: x / y
}

def lambda_handler(event, context):
    ""
    #
    """
```
Accepts an action and two numbers, performs the specified action on the numbers, and returns the result.

:param event: The event dict that contains the parameters sent when the function is invoked.
:param context: The context in which the function is called.
:return: The result of the specified action.

```
# Set the log level based on a variable configured in the Lambda environment.
logger.setLevel(os.environ.get('LOG_LEVEL', logging.INFO))
logger.debug('Event: %s', event)

action = event.get('action')
func = ACTIONS.get(action)
x = event.get('x')
y = event.get('y')
result = None
try:
    if func is not None and x is not None and y is not None:
        result = func(x, y)
        logger.info("%s %s %s is %s", x, action, y, result)
    else:
        logger.error("I can't calculate %s %s %s.", x, action, y)
except ZeroDivisionError:
    logger.warning("I can't divide %s by 0!", x)

response = {'result': result}
return response
```

- For API details, see the following topics in *AWS SDK for Go API Reference*.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration
Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-code-examples).

```java
/*
 * Lambda function names appear as:
 *
 *
 * To find this value, look at the function in the AWS Management Console.
 *
 * Before running this Java code example, set up your development environment, including your credentials.
 *
 * For more information, see this documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
 *
 * This example performs the following tasks:
 *
 * 1. Creates an AWS Lambda function.
 * 2. Gets a specific AWS Lambda function.
 * 3. Lists all Lambda functions.
 * 4. Invokes a Lambda function.
 * 5. Updates the Lambda function code and invokes it again.
 * 6. Updates a Lambda function's configuration value.
 * 7. Deletes a Lambda function.
 */

public class LambdaScenario {
    public static final String DASHES = new String(new char[80]).replace("\0", "-");  
    public static void main(String[] args) throws InterruptedException {
        final String usage = """
Usage:
   <functionName> <filePath> <role> <handler> <bucketName> <key>

Where:
  functionName - The name of the Lambda function.
  filePath - The path to the .zip or .jar where the code is located.
  role - The AWS Identity and Access Management (IAM) service role that has Lambda permissions.
  handler - The fully qualified method name (for example, example.Handler::handleRequest).
  bucketName - The Amazon Simple Storage Service (Amazon S3) bucket name that contains the .zip or .jar used to update the Lambda function's code.
  key - The Amazon S3 key name that represents the .zip or .jar (for example, LambdaHello-1.0-1.0-SNAPSHOT.jar).

if (args.length != 6) {
   System.out.println(usage);
   System.exit(1);
}

String functionName = args[0];
String filePath = args[1];
String role = args[2];
String handler = args[3];
String bucketName = args[4];
String key = args[5];

Region region = Region.US_WEST_2;
LambdaClient awsLambda = LambdaClient.builder()
   .region(region)
   .build();

System.out.println(DASHES);
System.out.println("Welcome to the AWS Lambda example scenario.");
System.out.println(DASHES);

System.out.println(DASHES);
System.out.println("1. Create an AWS Lambda function.");
String funArn = createLambdaFunction(awsLambda, functionName, filePath, role, handler);
System.out.println("The AWS Lambda ARN is "+funArn);
System.out.println(DASHES);
public static String createLambdaFunction(LambdaClient awsLambda,
public static void getFunction(LambdaClient awsLambda, String functionName) {
    try {
        LambdaWaiter waiter = awsLambda.waiter();
        InputStream is = new FileInputStream(filePath);
        SdkBytes fileToUpload = SdkBytes.fromInputStream(is);

        FunctionCode code = FunctionCode.builder()
            .zipFile(fileToUpload)
            .build();

        CreateFunctionRequest functionRequest =
            CreateFunctionRequest.builder()
                .functionName(functionName)
                .description("Created by the Lambda Java API")
                .code(code)
                .handler(handler)
                .runtime(Runtime.JAVA8)
                .role(role)
                .build();

        // Create a Lambda function using a waiter
        CreateFunctionResponse functionResponse =
            awsLambda.createFunction(functionRequest);
        GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
            .functionName(functionName)
            .build();
        WaiterResponse<GetFunctionResponse> waiterResponse =
            waiter.waitUntilFunctionExists(getFunctionRequest);
        waiterResponse.matched().response().ifPresent(System.out::println);
        return functionResponse.functionArn();
    } catch(LambdaException | FileNotFoundException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
    return "";
}
Get started with functions

public static void listFunctions(LambdaClient awsLambda) {
    try {
        ListFunctionsResponse functionResult = awsLambda.listFunctions();
        List<FunctionConfiguration> list = functionResult.functions();
        for (FunctionConfiguration config: list) {
            System.out.println("The function name is "+config.functionName());
        }
    }
    catch(LambdaException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}

public static void invokeFunction(LambdaClient awsLambda, String functionName) {
    InvokeResponse res;
    try {
        // Need a SdkBytes instance for the payload.
        JSONObject jsonObj = new JSONObject();
        jsonObj.put("inputValue", "2000");
        String json = jsonObj.toString();
        SdkBytes payload = SdkBytes.fromUtf8String(json) ;

        InvokeRequest request = InvokeRequest.builder()
               .functionName(functionName)
               .payload(payload)
               .build();

        InvokeResponse response =
               awsLambda.invoke(request);
        System.out.println("The response is "+response);
    }
    catch(LambdaException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}
.build();

res = awsLambda.invoke(request);
String value = res.payload().asUtf8String();
System.out.println(value);

} catch(LambdaException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}

public static void updateFunctionCode(LambdaClient awsLambda, String functionName, String bucketName, String key) {
    try {
        LambdaWaiter waiter = awsLambda.waiter();
        UpdateFunctionCodeRequest functionCodeRequest =
            UpdateFunctionCodeRequest.builder()
            .functionName(functionName)
            .publish(true)
            .s3Bucket(bucketName)
            .s3Key(key)
            .build();

        UpdateFunctionCodeResponse response = awsLambda.updateFunctionCode(functionCodeRequest);
        GetFunctionConfigurationRequest getFunctionConfigRequest =
            GetFunctionConfigurationRequest.builder()
            .functionName(functionName)
            .build();

        WaiterResponse<GetFunctionConfigurationResponse> waiterResponse = waiter.waitUntilFunctionUpdated(getFunctionConfigRequest);
        waiterResponse.matched().response().ifPresent(System.out::println);
        System.out.println("The last modified value is "+response.lastModified());

    } catch(LambdaException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}
public static void updateFunctionConfiguration(LambdaClient awsLambda, String functionName, String handler ){
    try {
        UpdateFunctionConfigurationRequest configurationRequest = UpdateFunctionConfigurationRequest.builder()
            .functionName(functionName)
            .handler(handler)
            .runtime(Runtime.JAVA11)
            .build();

        awsLambda.updateFunctionConfiguration(configurationRequest);
    } catch(LambdaException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}

public static void deleteLambdaFunction(LambdaClient awsLambda, String functionName) {
    try {
        DeleteFunctionRequest request = DeleteFunctionRequest.builder()
            .functionName(functionName)
            .build();

        awsLambda.deleteFunction(request);
        System.out.println("The "+functionName +" function was deleted");
    } catch(LambdaException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}

For API details, see the following topics in AWS SDK for Java 2.x API Reference.

- [CreateFunction](#)
- [DeleteFunction](#)
- [GetFunction](#)
- [Invoke](#)
• **ListFunctions**
• **UpdateFunctionCode**
• **UpdateFunctionConfiguration**

**JavaScript**

**SDK for JavaScript (v3)**

```javascript
Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
```

Create an AWS Identity and Access Management (IAM) role that grants Lambda permission to write to logs.

```javascript
log(`Creating role (${NAME_ROLE_LAMBDA})...`);
const response = await createRole(NAME_ROLE_LAMBDA);

import { AttachRolePolicyCommand, IAMClient } from '@aws-sdk/client-iam';

const client = new IAMClient({});

/**
 * @param {string} policyArn
 * @param {string} roleName
 */
export const attachRolePolicy = (policyArn, roleName) => {
    const command = new AttachRolePolicyCommand({
        PolicyArn: policyArn,
        RoleName: roleName,
    });

    return client.send(command);
};
```

Create a Lambda function and upload handler code.
const createFunction = async (funcName, roleArn) => {
    const client = new LambdaClient({});
    const code = await readFile(`${dirname}../functions/${funcName}.zip`);

    const command = new CreateFunctionCommand({
        Code: { ZipFile: code },
        FunctionName: funcName,
        Role: roleArn,
        Architectures: [Architecture.arm64],
        Handler: "index.handler", // Required when sending a .zip file
        PackageType: PackageType.Zip, // Required when sending a .zip file
        Runtime: Runtime.nodejs16x, // Required when sending a .zip file
    });

    return client.send(command);
};

Invoke the function with a single parameter and get results.

const invoke = async (funcName, payload) => {
    const client = new LambdaClient({});
    const command = new InvokeCommand({
        FunctionName: funcName,
        Payload: JSON.stringify(payload),
        LogType: LogType.Tail,
    });

    const { Payload, LogResult } = await client.send(command);
    const result = Buffer.from(Payload).toString();
    const logs = Buffer.from(LogResult, "base64").toString();
    return { logs, result };
};

Update the function code and configure its Lambda environment with an environment variable.

const updateFunctionCode = async (funcName, newFunc) => {
    const client = new LambdaClient({});
    const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
    const command = new UpdateFunctionCodeCommand({}
ZipFile: code,
FunctionName: funcName,
Architectures: [Architecture.arm64],
Handler: "index.handler", // Required when sending a .zip file
PackageType: PackageType.Zip, // Required when sending a .zip file
Runtime: Runtime.nodejs16x, // Required when sending a .zip file
});

return client.send(command);
};

const updateFunctionConfiguration = (funcName) => {
    const client = new LambdaClient({});
    const config = readFileSync(`${dirname}../functions/config.json`).toString();
    const command = new UpdateFunctionConfigurationCommand({
        ...JSON.parse(config),
        FunctionName: funcName,
    });
    return client.send(command);
};

List the functions for your account.

const listFunctions = () => {
    const client = new LambdaClient({});
    const command = new ListFunctionsCommand({});

    return client.send(command);
};

Delete the IAM role and the Lambda function.

import { DeleteRoleCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 * @param {string} roleName
 */
export const deleteRole = (roleName) => {

const command = new DeleteRoleCommand({ RoleName: roleName });
  return client.send(command);
};

/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

**Kotlin**

**SDK for Kotlin**

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

suspend fun main(args: Array<String>) {
  val usage = ""
  Usage:
<functionName> <role> <handler> <bucketName> <updatedBucketName> <key>

Where:
functionName - The name of the AWS Lambda function.
    role - The AWS Identity and Access Management (IAM) service role that has AWS Lambda permissions.
    handler - The fully qualified method name (for example, example.Handler::handleRequest).
    bucketName - The Amazon Simple Storage Service (Amazon S3) bucket name that contains the ZIP or JAR used for the Lambda function's code.
    updatedBucketName - The Amazon S3 bucket name that contains the .zip or .jar used to update the Lambda function's code.
    key - The Amazon S3 key name that represents the .zip or .jar file (for example, LambdaHello-1.0-SNAPSHOT.jar).

if (args.size != 6) {
    println(usage)
    exitProcess(1)
}

val functionName = args[0]
val role = args[1]
val handler = args[2]
val bucketName = args[3]
val updatedBucketName = args[4]
val key = args[5]

println("Creating a Lambda function named \\
    \$\{functionName\}.")
val funArn = createScFunction(functionName, bucketName, key, handler, role)
println("The AWS Lambda ARN is \$\{funArn\}"

// Get a specific Lambda function.
println("Getting the \$\{functionName\} AWS Lambda function.")
getFunction(functionName)

// List the Lambda functions.
println("Listing all AWS Lambda functions.")
listFunctionsSc()

// Invoke the Lambda function.
println("*** Invoke the Lambda function.")
invokeFunctionSc(functionName)
// Update the AWS Lambda function code.
println("*** Update the Lambda function code.")
updateFunctionCode(functionName, updatedBucketName, key)

// println("*** Invoke the function again after updating the code.")
invokeFunctionSc(functionName)

// Update the AWS Lambda function configuration.
println("Update the run time of the function.")
UpdateFunctionConfiguration(functionName, handler)

// Delete the AWS Lambda function.
println("Delete the AWS Lambda function.")
delFunction(functionName)
}
suspend fun createScFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String
): String {

    val functionCode = FunctionCode {
        s3Bucket = s3BucketName
        s3Key = myS3Key
    }

    val request = CreateFunctionRequest {
        functionName = myFunctionName
        code = functionCode
        description = "Created by the Lambda Kotlin API"
        handler = myHandler
        role = myRole
        runtime = Runtime.Java8
    }

    // Create a Lambda function using a waiter
    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
suspend fun getFunction(functionNameVal: String) {
    val functionRequest = GetFunctionRequest {
        functionName = functionNameVal
    }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val response = awsLambda.getFunction(functionRequest)
        println("The runtime of this Lambda function is
        \${response.configuration?.runtime}"
        )
    }
}

suspend fun listFunctionsSc() {
    val request = ListFunctionsRequest {
        maxItems = 10
    }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val response = awsLambda.listFunctions(request)
        response.functions?.forEach { function ->
            println("The function name is \${function.functionName}"
        }
    }
}

suspend fun invokeFunctionSc(functionNameVal: String) {
    val json = """{"inputValue":"1000"}""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request = InvokeRequest {
        functionName = functionNameVal
        payload = byteArray
        logType = LogType.Tail
    }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
println("The function payload is \n${res.payload?.toString(Charsets.UTF_8)}")

suspend fun updateFunctionCode(functionNameVal: String?, bucketName: String?, key: String?) {

    val functionCodeRequest = UpdateFunctionCodeRequest {
        functionName = functionNameVal
        publish = true
        s3Bucket = bucketName
        s3Key = key
    }

    LambdaClient { region = "us-west-2" }.use { awslambda ->
        val response = awslambda.updateFunctionCode(functionCodeRequest)
        awslambda.waitUntilFunctionUpdated {
            functionName = functionNameVal
        }
        println("The last modified value is " + response.lastModified)
    }
}

suspend fun UpdateFunctionConfiguration(functionNameVal: String?, handlerVal: String?) {

    val configurationRequest = UpdateFunctionConfigurationRequest {
        functionName = functionNameVal
        handler = handlerVal
        runtime = Runtime.Java11
    }

    LambdaClient { region = "us-west-2" }.use { awslambda ->
        awslambda.updateFunctionConfiguration(configurationRequest)
    }
}

suspend fun delFunction(myFunctionName: String) {

    val request = DeleteFunctionRequest {
        functionName = myFunctionName
    }
}
For API details, see the following topics in *AWS SDK for Kotlin API reference*.

- [CreateFunction](#)
- [DeleteFunction](#)
- [GetFunction](#)
- [Invoke](#)
- [ListFunctions](#)
- [UpdateFunctionCode](#)
- [UpdateFunctionConfiguration](#)

### PHP

**SDK for PHP**

```php
namespace Lambda;

use Aws\S3\S3Client;
use GuzzleHttp\Psr7\Stream;
use Iam\IAMService;

class GettingStartedWithLambda
{
    public function run()
    {
        echo("\n");
        echo("---------------------\n");
    }
}
```

---

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-php).

```php
namespace Lambda;

use Aws\S3\S3Client;
use GuzzleHttp\Psr7\Stream;
use Iam\IAMService;

class GettingStartedWithLambda
{
    public function run()
    {
        echo("\n");
        echo("---------------------\n");
    }
}
```
print("Welcome to the AWS Lambda getting started demo using PHP!\n");
echo("--------------------------------------\n");

$clientArgs = [
    'region' => 'us-west-2',
    'version' => 'latest',
    'profile' => 'default',
];
$uniqid = uniqid();

$iamService = new IAMService();
$s3client = new S3Client($clientArgs);
$lambdaService = new LambdaService();

echo "First, let's create a role to run our Lambda code.\n";
$roleName = "test-lambda-role-$uniqid";
$rolePolicyDocument = "{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
";
$role = $iamService->createRole($roleName, $rolePolicyDocument);
echo "Created role {$role['RoleName']}.
"

$iamService->attachRolePolicy($role['RoleName'], "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole");
echo "Attached the AWSLambdaBasicExecutionRole to {$role['RoleName']}.\n"

echo "\nNow let's create an S3 bucket and upload our Lambda code there.\n";
$bucketName = "test-example-bucket-$uniqid";
$s3client->createBucket([ 'Bucket' => $bucketName, ]);
echo "Created bucket $bucketName.\n";

$functionName = "doc_example_lambda_$uniqid";
$codeBasic = __DIR__. "/lambda_handler_basic.zip";
$handler = "lambda_handler_basic";
$file = file_get_contents($codeBasic);
$s3client->putObject(
    'Bucket' => $bucketName,
    'Key' => $functionName,
    'Body' => $file,
);
echo "Uploaded the Lambda code.\n";

$createLambdaFunction = $lambdaService->createFunction($functionName, $role, $bucketName, $handler);
// Wait until the function has finished being created.
do {
    $getLambdaFunction = $lambdaService->getFunction($createLambdaFunction['FunctionName']);
} while ($getLambdaFunction['Configuration']['State'] == "Pending");
echo "Created Lambda function {$getLambdaFunction['Configuration']['FunctionName']}.\n";

sleep(1);

echo "\nOk, let's invoke that Lambda code.\n";
$basicParams = [
    'action' => 'increment',
    'number' => 3,
];
/** @var Stream $invokeFunction */
$invokeFunction = $lambdaService->invoke($functionName, $basicParams)['Payload'];
$result = json_decode($invokeFunction->getContents())->result;
echo "After invoking the Lambda code with the input of {$basicParams['number']} we received $result.\n";

echo "\nSince that's working, let's update the Lambda code.\n";
$codeCalculator = "lambda_handler_calculator.zip";
$handlerCalculator = "lambda_handler_calculator";
echo "First, put the new code into the S3 bucket.\n";
$file = file_get_contents($codeCalculator);
$s3client->putObject([  
    'Bucket' => $bucketName,  
    'Key' => $functionName,  
    'Body' => $file,  
]);

Get started with functions
'Key' => $functionName,
'Bdy' => $file,
]);
echo "New code uploaded."

$lambdaService->updateFunctionCode($functionName, $bucketName, $functionName);
// Wait for the Lambda code to finish updating.
do {
    $getLambdaFunction = $lambdaService->getFunction($createLambdaFunction['FunctionName']);
} while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !== "Successful");
echo "New Lambda code uploaded."

$environment = [
    'Variable' => ['Variables' => ['LOG_LEVEL' => 'DEBUG']]
];
$lambdaService->updateFunctionConfiguration($functionName, $handlerCalculator, $environment);
do {
    $getLambdaFunction = $lambdaService->getFunction($createLambdaFunction['FunctionName']);
} while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !== "Successful");
echo "Lambda code updated with new handler and a LOG_LEVEL of DEBUG for more information."

echo "Invoke the new code with some new data."
$calculatorParams = [
    'action' => 'plus',
    'x' => 5,
    'y' => 4,
];
$invokeFunction = $lambdaService->invoke($functionName, $calculatorParams, "Tail");
$result = json_decode($invokeFunction['Payload']->getContents())->result;
echo "Indeed, {$calculatorParams['x']} + {$calculatorParams['y']} does equal $result."
echo "Here's the extra debug info: ";
echo base64_decode($invokeFunction['LogResult']). "\n"

echo "But what happens if you try to divide by zero?"
$divZeroParams = [
'action' => 'divide',
'x' => 5,
'y' => 0,
];
$invokeFunction = $lambdaService->invoke($functionName, $divZeroParams,
"Tail");
$result = json_decode($invokeFunction['Payload']->getContents())->result;
echo "You get a |$result| result.\n";
echo "And an error message: ";
echo base64_decode($invokeFunction['LogResult']) . "\n";
echo "\nHere's all the Lambda functions you have in this Region:\n";
$listLambdaFunctions = $lambdaService->listFunctions(5);
$allLambdaFunctions = $listLambdaFunctions['Functions'];
$next = $listLambdaFunctions->get('NextMarker');
while ($next !== false) {
    $listLambdaFunctions = $lambdaService->listFunctions(5, $next);
    $next = $listLambdaFunctions->get('NextMarker');
    $allLambdaFunctions = array_merge($allLambdaFunctions,
        $listLambdaFunctions['Functions']);
}
foreach ($allLambdaFunctions as $function) {
    echo "{$function['FunctionName']}\n";
}
echo "\nAnd don't forget to clean up your data!\n";

$lambdaService->deleteFunction($functionName);
echo "Deleted Lambda function.\n";
$iamService->deleteRole($role['RoleName']);
echo "Deleted Role.\n";
$deleteObjects = $s3client->listObjectsV2([ 
    'Bucket' => $bucketName,
]);
$deleteObjects = $s3client->deleteObjects([ 
    'Bucket' => $bucketName,
    'Delete' => [
        'Objects' => $deleteObjects['Contents'],
    ]
]);
echo "Deleted all objects from the S3 bucket.\n";
$s3client->deleteBucket([ 'Bucket' => $bucketName]);
echo "Deleted the bucket.\n";
• For API details, see the following topics in *AWS SDK for PHP API Reference*.
  • [CreateFunction](https://aws.amazon.com/)
  • [DeleteFunction](https://aws.amazon.com/)
  • [GetFunction](https://aws.amazon.com/)
  • [Invoke](https://aws.amazon.com/)
  • [ListFunctions](https://aws.amazon.com/)
  • [UpdateFunctionCode](https://aws.amazon.com/)
  • [UpdateFunctionConfiguration](https://aws.amazon.com/)

**Python**

**SDK for Python (Boto3)**

---

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://aws.amazon.com/).

---

Define a Lambda handler that increments a number.

```python
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    
    Accepts an action and a single number, performs the specified action on the number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the function
                  is invoked.
```
Define a second Lambda handler that performs arithmetic operations.

```python
import logging
import os

logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}

def lambda_handler(event, context):
    ""
    Accepts an action and two numbers, performs the specified action on the numbers, and returns the result.
    :param event: The event dict that contains the parameters sent when the function
is invoked.
:param context: The context in which the function is called.
:return: The result of the specified action.

# Set the log level based on a variable configured in the Lambda environment.
logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
logger.debug("Event: %s", event)

action = event.get("action")
func = ACTIONS.get(action)
x = event.get("x")
y = event.get("y")
result = None
try:
    if func is not None and x is not None and y is not None:
        result = func(x, y)
        logger.info("%s %s %s is %s", x, action, y, result)
    else:
        logger.error("I can't calculate %s %s %s.", x, action, y)
except ZeroDivisionError:
    logger.warning("I can't divide %s by 0!", x)

response = {"result": result}
return response

Create functions that wrap Lambda actions.

class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource

@staticmethod
    def create_deployment_package(source_file, destination_file):
        """
        Creates a Lambda deployment package in .zip format in an in-memory
        buffer. This
        buffer can be passed directly to Lambda when creating the function.
        """
buffer = io.BytesIO()
with zipfile.ZipFile(buffer, "w") as zipped:
    zipped.write(source_file, destination_file)
buffer.seek(0)
return buffer.read()

def get_iam_role(self, iam_role_name):
    ""
    Get an AWS Identity and Access Management (IAM) role.
    ""
    role = None
    try:
        temp_role = self.iam_resource.Role(iam_role_name)
        temp_role.load()
        role = temp_role
        logger.info("Got IAM role %s", role.name)
    except ClientError as err:
        if err.response["Error"]["Code"] == "NoSuchEntity":
            logger.info("IAM role %s does not exist.", iam_role_name)
        else:
            logger.error(          "Couldn't get IAM role %s. Here's why: %s: %s",
            iam_role_name,          err.response["Error"]["Code"],          err.response["Error"]["Message"],          )
        raise
    return role

def create_iam_role_for_lambda(self, iam_role_name):
    ""
    Creates an IAM role that grants the Lambda function basic permissions. If a
    role with the specified name already exists, it is used for the demo.
:param iam_role_name: The name of the role to create.
:return: The role and a value that indicates whether the role is newly created.

```python
role = self.get_iam_role(iam_role_name)
if role is not None:
    return role, False

lambda_assume_role_policy = {
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {"Service": "lambda.amazonaws.com"},
            "Action": "sts:AssumeRole",
        }
    ],
}

policy_arn = "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"

try:
    role = self.iam_resource.create_role(
        RoleName=iam_role_name,
        AssumeRolePolicyDocument=json.dumps(lambda_assume_role_policy),
    )
    logger.info("Created role %s.", role.name)
    role.attach_policy(PolicyArn=policy_arn)
    logger.info("Attached basic execution policy to role %s.", role.name)
except ClientError as error:
    if error.response["Error"]["Code"] == "EntityAlreadyExists":
        role = self.iam_resource.Role(iam_role_name)
        logger.warning("The role %s already exists. Using it.",
            iam_role_name)
    else:
        logger.exception("Couldn't create role %s or attach policy %s.",
            iam_role_name,
            policy_arn,
        )
        raise

return role, True
```
def get_function(self, function_name):
    """
    Gets data about a Lambda function.
    
    :param function_name: The name of the function.
    :return: The function data.
    """
    response = None
    try:
        response = self.lambda_client.get_function(FunctionName=function_name)
    except ClientError as err:
        if err.response['Error']['Code'] == 'ResourceNotFoundException':
            logger.info("Function %s does not exist.", function_name)
        else:
            logger.error("Couldn't get function %s. Here's why: %s: %s",
                         function_name,
                         err.response['Error']['Code'],
                         err.response['Error']['Message'],
                     )
            raise
    return response

def create_function(self, function_name, handler_name, iam_role, deployment_package):
    """
    Deploys a Lambda function.
    
    :param function_name: The name of the Lambda function.
    :param handler_name: The fully qualified name of the handler function. This
                         must include the file name and the function name.
    :param iam_role: The IAM role to use for the function.
    :param deployment_package: The deployment package that contains the function
                               code in .zip format.
    :return: The Amazon Resource Name (ARN) of the newly created function.
    """
    try:
        response = self.lambda_client.create_function(
            FunctionName=function_name,
            Handler=handler_name,
            Role=iam_role,
            Code=os.path.abspath(deployment_package)
        )
    except ClientError as err:
        if err.response['Error']['Code'] == 'ResourceNotFoundException':
            logger.info("Function %s does not exist.", function_name)
        else:
            logger.error("Couldn't create function %s. Here's why: %s: %s",
                         function_name,
                         err.response['Error']['Code'],
                         err.response['Error']['Message'],
                     )
            raise
    return response
def delete_function(self, function_name):
    ""
    Deletes a Lambda function.
    
    :param function_name: The name of the function to delete.
    ""
    try:
        self.lambda_client.delete_function(FunctionName=function_name)
    except ClientError:
        logger.exception("Couldn't delete function %s.", function_name)
    raise

def invoke_function(self, function_name, function_params, get_log=False):
    ""
    Invokes a Lambda function.
    
    :param function_name: The name of the function to invoke.
    :param function_params: The parameters of the function as a dict. This dict
    is serialized to JSON before it is sent to Lambda.
:param get_log: When true, the last 4 KB of the execution log are included in the response.
:return: The response from the function invocation.

```
try:
    response = self.lambda_client.invoke(
        FunctionName=function_name,
        Payload=json.dumps(function_params),
        LogType="Tail" if get_log else "None",
    )
    logger.info("Invoked function %s.", function_name)
except ClientError:
    logger.exception("Couldn't invoke function %s.", function_name)
    raise
return response
```

def update_function_code(self, function_name, deployment_package):
    ""
    Updates the code for a Lambda function by submitting a .zip archive that contains the code for the function.
    
    :param function_name: The name of the function to update.
    :param deployment_package: The function code to update, packaged as bytes in .zip format.
    :return: Data about the update, including the status.
    ""
    try:
        response = self.lambda_client.update_function_code(
            FunctionName=function_name, ZipFile=deployment_package
        )
    except ClientError as err:
        logger.error(
            "Couldn't update function %s. Here's why: %s: %s",
            function_name,
            err.response["Error"]['Code'],
            err.response["Error"]['Message'],
        )
        raise
    else:
        return response
def update_function_configuration(self, function_name, env_vars):
    
    Updates the environment variables for a Lambda function.
    
    :param function_name: The name of the function to update.
    :param env_vars: A dict of environment variables to update.
    :return: Data about the update, including the status.
    
    try:
        response = self.lambda_client.update_function_configuration(
            FunctionName=function_name, Environment={"Variables": env_vars}
        )
    except ClientError as err:
        logger.error("Couldn't update function configuration \s. Here’s why: %s: %s",
                      function_name,
                      err.response["Error"]["Code"],
                      err.response["Error"]["Message"],
                  )
        raise
    else:
        return response

def list_functions(self):

    Lists the Lambda functions for the current account.

    try:
        funcPaginator = self.lambda_client.get_paginator("list_functions")
        for funcPage in funcPaginator.paginate():
            for func in funcPage["Functions"]:
                print(func["FunctionName"])
                desc = func.get("Description")
                if desc:
                    print(f"\t{desc}")
                    print(f"\t{func["Runtime"]}: {func["Handler"]}"))
    except ClientError as err:
        logger.error("Couldn't list functions. Here's why: %s: %s",
                      err.response["Error"]["Code"],
                      err.response["Error"]["Message"],
                  )
Create a function that runs the scenario.

class UpdateFunctionWaiter(CustomWaiter):
    """A custom waiter that waits until a function is successfully updated."""
    def __init__(self, client):
        super().__init__(
            "UpdateSuccess",
            "GetFunction",
            "Configuration.LastUpdateStatus",
            {"Successful": WaitState.SUCCESS, "Failed": WaitState.FAILURE},
            client,
        )

    def wait(self, function_name):
        self._wait(FunctionName=function_name)

def run_scenario(lambda_client, iam_resource, basic_file, calculator_file, lambda_name):
    """
    Runs the scenario.

    :param iam_resource: A Boto3 IAM resource.
    :param basic_file: The name of the file that contains the basic Lambda handler.
    :param calculator_file: The name of the file that contains the calculator Lambda handler.
    :param lambda_name: The name to give resources created for the scenario, such as the
                         IAM role and the Lambda function.
    """
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")
    print("-" * 88)
print("Welcome to the AWS Lambda getting started with functions demo.")
print("-" * 88)

wrapper = LambdaWrapper(lambda_client, iam_resource)

print("Checking for IAM role for Lambda...")
iam_role, should_wait = wrapper.create_iam_role_for_lambda(lambda_name)
if should_wait:
    logger.info("Giving AWS time to create resources...")
    wait(10)

print(f"Looking for function {lambda_name}...")
function = wrapper.get_function(lambda_name)
if function is None:
    print("Zipping the Python script into a deployment package...")
    deployment_package = wrapper.create_deployment_package(
        basic_file, f"{lambda_name}.py"
    )
    print(f"...and creating the {lambda_name} Lambda function.")
    wrapper.create_function(
        lambda_name, f"{lambda_name}.lambda_handler", iam_role,
        deployment_package
    )
else:
    print(f"Function {lambda_name} already exists.")
print("-" * 88)

print(f"Let's invoke {lambda_name}. This function increments a number.")
action_params = {
    "action": "increment",
    "number": q.ask("Give me a number to increment: ", q.is_int),
}
print(f"Invoking {lambda_name}...")
response = wrapper.invoke_function(lambda_name, action_params)
print(  
    f"Incrementing [action_params['number']] resulted in "
    f"[json.load(response['Payload'])]"
)
print("-" * 88)

print(f"Let's update the function to an arithmetic calculator.")
q.ask("Press Enter when you're ready.")
print("Creating a new deployment package...")
 deployment_package = wrapper.create_deployment_package(
calculator_file, f"{lambda_name}.py"
)
print(f"...and updating the {lambda_name} Lambda function.")
update_waiter = UpdateFunctionWaiter(lambda_client)
wrapper.update_function_code(lambda_name, deployment_package)
update_waiter.wait(lambda_name)
print(f"This function uses an environment variable to control logging level.")
print(f"Let's set it to DEBUG to get the most logging.")
wrapper.update_function_configuration(
    lambda_name, {"LOG_LEVEL": logging.getLevelName(logging.DEBUG)}
)

actions = ["plus", "minus", "times", "divided-by"]
want_invoke = True
while want_invoke:
    print(f"Let's invoke {lambda_name}. You can invoke these actions:")
    for index, action in enumerate(actions):
        print(f"{index + 1}: {action}")
    action_params = {}
    action_index = q.ask(
        "Enter the number of the action you want to take: ",
        q.is_int,
        q.in_range(1, len(actions)),
    )
    action_params["action"] = actions[action_index - 1]
    print(f"You've chosen to invoke 'x {action_params["action"]} y'.")
    action_params["x"] = q.ask("Enter a value for x: ", q.is_int)
    action_params["y"] = q.ask("Enter a value for y: ", q.is_int)
    print(f"Invoking {lambda_name}...")
    response = wrapper.invoke_function(lambda_name, action_params, True)
    print(f"Calculating {action_params["x"]} {action_params["action"]} {action_params["y"]} resulted in {json.load(response['Payload'])}"
    )
    q.ask("Press Enter to see the logs from the call.")
    print(base64.b64decode(response["LogResult"]).decode())
    want_invoke = q.ask("That was fun. Shall we do it again? (y/n) ",
        q.is_yesno)
    print("-" * 88)

if q.ask("Get started with functions 1919")
"Do you want to list all of the functions in your account? (y/n) ", q.is_yeson
wrapper.list_functions()
print("-" * 88)

if q.ask("Ready to delete the function and role? (y/n) ", q.is_yeson):
    for policy in iam_role.attached_policies.all():
        policy.detach_role(RoleName=iam_role.name)
    iam_role.delete()
    print(f"Deleted role {lambda_name}.")
    wrapper.delete_function(lambda_name)
    print(f"Deleted function {lambda_name}.")

print("\nThanks for watching!")
print("-" * 88)

if __name__ == "__main__":
    try:
        run_scenario(
            boto3.client("lambda"),
            boto3.resource("iam"),
            "lambda_handler_basic.py",
            "lambda_handler_calculator.py",
            "doc_example_lambda_calculator",
        )
    except Exception:
        logging.exception("Something went wrong with the demo!")
Set up pre-requisite IAM permissions for a Lambda function capable of writing logs.

```ruby
# Get an AWS Identity and Access Management (IAM) role.
#
# @param iam_role_name: The name of the role to retrieve.
# @param action: Whether to create or destroy the IAM apparatus.
# @return: The IAM role.
def manage_iam(iam_role_name, action)
    role_policy = {
        'Version': "2012-10-17",
        'Statement': [
            {
                'Effect': "Allow",
                'Principal': {
                    'Service': "lambda.amazonaws.com"
                },
                'Action': "sts:AssumeRole"
            }
        ]
    }
    case action
    when "create"
        role = $iam_client.create_role(
            role_name: iam_role_name,
            assume_role_policy_document: role_policy.to_json
        )
        $iam_client.attach_role_policy(
            policy_arn: "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole",
            role_name: iam_role_name
        )
    end
```

Note: There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-code-examples).
Define a Lambda handler that increments a number provided as an invocation parameter.

```ruby
require "logger"

# A function that increments a whole number by one (1) and logs the result.
# Requires a manually-provided runtime parameter, 'number', which must be Int
#
# @param event [Hash] Parameters sent when the function is invoked
# @param context [Hash] Methods and properties that provide information
# about the invocation, function, and execution environment.
# @return incremented_number [String] The incremented number.
def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
```
log_level = ENV['LOG_LEVEL']
logger.level = case log_level
    when "debug"
        Logger::DEBUG
    when "info"
        Logger::INFO
    else
        Logger::ERROR
end
logger.debug("This is a debug log message.")
logger.info("This is an info log message. Code executed successfully!")
number = event['number'].to_i
incremented_number = number + 1
logger.info("You provided #{number.round} and it was incremented to
#{incremented_number.round}")
incremented_number.round.to_s
end

Zip your Lambda function into a deployment package.

```ruby
# Creates a Lambda deployment package in .zip format.
# This zip can be passed directly as a string to Lambda when creating the function.
#
# @param source_file: The name of the object, without suffix, for the Lambda file and zip.
# @return: The deployment package.

def create_deployment_package(source_file)
    Dir.chdir(File.dirname(__FILE__))
    if File.exist?('lambda_function.zip')
        File.delete('lambda_function.zip')
        @logger.debug("Deleting old zip: lambda_function.zip")
    end
    Zip::File.open('lambda_function.zip', create: true) do |zipfile|
        zipfile.add("lambda_function.rb", "#{source_file}.rb")
    end
    @logger.debug("Zipping #{source_file}.rb into: lambda_function.zip.")
    File.read("lambda_function.zip").to_s
    rescue StandardError => e
        @logger.error("There was an error creating deployment package:
#{e.message}")
```
Create a new Lambda function.

```ruby
# Deploys a Lambda function.
#
# @param function_name: The name of the Lambda function.
# @param handler_name: The fully qualified name of the handler function. This
#   must include the file name and the function name.
# @param role_arn: The IAM role to use for the function.
# @param deployment_package: The deployment package that contains the function
code in .zip format.
# @return: The Amazon Resource Name (ARN) of the newly created function.

def create_function(function_name, handler_name, role_arn, deployment_package)
    response = @lambda_client.create_function({
        role: role_arn.to_s,
        function_name: function_name,
        handler: handler_name,
        runtime: "ruby2.7",
        code: {
            zip_file: deployment_package
        },
        environment: {
            variables: {
                "LOG_LEVEL" => "info"
            }
        }
    })
    @lambda_client.wait_until(:function_active_v2, { function_name: function_name}) do |w|
        w.max_attempts = 5
        w.delay = 5
    end
    response
rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error creating #{function_name}:
#{e.message}"
rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:
#{e.message}"
end
```

Get started with functions
Invoke your Lambda function with optional runtime parameters.

```ruby
# Invokes a Lambda function.
# @param function_name [String] The name of the function to invoke.
# @param payload [nil] Payload containing runtime parameters.
# @return [Object] The response from the function invocation.
def invoke_function(function_name, payload = nil)
  params = { function_name: function_name}
  params[:payload] = payload unless payload.nil?
  @lambda_client.invoke(params)
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:

    #{e.message}"
  end
end
```

Update your Lambda function's configuration to inject a new environment variable.

```ruby
# Updates the environment variables for a Lambda function.
# @param function_name: The name of the function to update.
# @param log_level: The log level of the function.
# @return: Data about the update, including the status.
def update_function_configuration(function_name, log_level)
  @lambda_client.update_function_configuration({
    function_name: function_name,
    environment: {
      variables: {
        "LOG_LEVEL" => log_level
      }
    }
  })
  @lambda_client.wait_until(:function_updated_v2, { function_name:
    function_name}) do |w|
    w.max_attempts = 5
    w.delay = 5
  end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating configurations for

    #{function_name}:

    #{e.message}"
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:

    #{e.message}"
  end
end
```
Update your Lambda function's code with a different deployment package containing different code.

```ruby
# Updates the code for a Lambda function by submitting a .zip archive that contains
# the code for the function.

# @param function_name: The name of the function to update.
# @param deployment_package: The function code to update, packaged as bytes in
# .zip format.
# @return: Data about the update, including the status.
def update_function_code(function_name, deployment_package)
    @lambda_client.update_function_code(  
        function_name: function_name,  
        zip_file: deployment_package
    )
    @lambda_client.wait_until(:function_updated_v2, {  
        function_name: function_name
    }) do |w|
        w.max_attempts = 5
        w.delay = 5
    end
    rescue Aws::Lambda::Errors::ServiceException => e
        @logger.error("There was an error updating function code for:
        #{function_name}:
        #{e.message}")
        nil
    rescue Aws::Waiters::Errors::WaiterFailed => e
        @logger.error("Failed waiting for #{function_name} to update:
        #{e.message}")
    end
```

List all existing Lambda functions using the built-in paginator.

```ruby
# Lists the Lambda functions for the current account.
def list_functions
    functions = []
    @lambda_client.list_functions.each do |response|
        response["functions"].each do |function|
            functions.append(function["function_name"])
        end
    end
end
```
functions
rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:
#{e.message}")
end

Delete a specific Lambda function.

# Deletes a Lambda function.
# @param function_name: The name of the function to delete.
def delete_function(function_name)
    print "Deleting function: #{function_name}..."
    @lambda_client.delete_function(
        function_name: function_name
    )
    print "Done!".green
rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error deleting #{function_name}:
#{e.message}")
end

• For API details, see the following topics in AWS SDK for Ruby API Reference.
  • CreateFunction
  • DeleteFunction
  • GetFunction
  • Invoke
  • ListFunctions
  • UpdateFunctionCode
  • UpdateFunctionConfiguration
Rust

SDK for Rust

**Note**
There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-code-examples).

The Cargo.toml with dependencies used in this scenario.

```toml
[package]
name = "lambda-code-examples"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
aws-config = { features = ["behavior-version-latest"], version = "1.0.1" }
aws-sdk-ec2 = "1.2.0"
aws-sdk-iam = "1.2.0"
aws-sdk-lambda = "1.2.0"
aws-sdk-s3 = "1.2.0"
aws-smithy-types = "1.0.1"
aws-types = { git = "https://github.com/awslabs/aws-sdk-rust", branch = "next" }
clap = { version = "~4.4", features = ["derive"] }
tokio = { version = "1.20.1", features = ["full"] }
tracing-subscriber = { version = "0.3.15", features = ["env-filter"] }
tracing = "0.1.37"
serde_json = "1.0.94"
anyhow = "1.0.71"
uuid = { version = "1.3.3", features = ["v4"] }
lambda_runtime = "0.8.0"
serde = "1.0.164"
```

A collection of utilities that streamline calling Lambda for this scenario. This file is src/inations.rs in the crate.
use anyhow::anyhow;
use aws_sdk_iam::operation::delete_role::DeleteRoleOutput;
use aws_sdk_lambda::{
    operation::{
        delete_function::DeleteFunctionOutput, get_function::GetFunctionOutput,
        invoke::InvokeOutput, list_functions::ListFunctionsOutput,
        update_function_code::UpdateFunctionCodeOutput,
        update_function_configuration::UpdateFunctionConfigurationOutput,
    },
    primitives::ByteStream,
    types::{Environment, FunctionCode, LastUpdateStatus, State},
};
use aws_sdk_s3::{
    operation::{delete_bucket::DeleteBucketOutput,
                delete_object::DeleteObjectOutput},
    types::CreateBucketConfiguration,
};
use aws_smithy_types::Blob;
use serde::{ser::SerializeMap, Serialize};
use std::{path::PathBuf, str::FromStr, time::Duration};
use tracing::{debug, info, warn};

/* Operation describes */
#[derive(Clone, Copy, Debug, Serialize)]
pub enum Operation {
    #[serde(rename = "plus")]
    Plus,
    #[serde(rename = "minus")]
    Minus,
    #[serde(rename = "times")]
    Times,
    #[serde(rename = "divided-by")]
    DividedBy,
}

impl FromStr for Operation {
    type Err = anyhow::Error;
}
fn from_str(s: &str) -> Result<Self, Self::Err> {
    match s {
        "plus" => Ok(Operation::Plus),
        "minus" => Ok(Operation::Minus),
        "times" => Ok(Operation::Times),
        "divided-by" => Ok(Operation::DividedBy),
        _ => Err(anyhow!("Unknown operation {s}")),
    }
}
}

impl ToString for Operation {
    fn to_string(&self) -> String {
        match self {
            Operation::Plus => "plus".to_string(),
            Operation::Minus => "minus".to_string(),
            Operation::Times => "times".to_string(),
            Operation::DividedBy => "divided-by".to_string(),
        }
    }
}

/**
 * InvokeArgs will be serialized as JSON and sent to the AWS Lambda handler.
 */
#[derive(Debug)]
pub enum InvokeArgs {
    Increment(i32),
    Arithmetic(Operation, i32, i32),
}

impl Serialize for InvokeArgs {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        match self {
            InvokeArgs::Increment(i) => serializer.serialize_i32(*i),
            InvokeArgs::Arithmetic(o, i, j) => {
                let mut map: S::SerializeMap = serializer.serialize_map(Some(3))?
                    .serialize_key(&"op".to_string())?
                    .serialize_value(&o.to_string())?
                    .serialize_key(&"i".to_string())?
                    .serialize_value(&i.to_string())?
                    .serialize_value(&j.to_string())?
            }
        }
    }
}
map.serialize_value(&i)?;
map.serialize_key(&"j".to_string())?;
map.serialize_value(&j)?;
map.end()
}
}
}
}

/** A policy document allowing Lambda to execute this function on the account's behalf. */
const ROLE_POLICY_DOCUMENT: &str = r#
"Version": "2012-10-17",
"Statement": [
    {
        "Effect": "Allow",
        "Principal": { "Service": "lambda.amazonaws.com" },
        "Action": "sts:AssumeRole"
    }
]
}"

/**
* A LambdaManager gathers all the resources necessary to run the Lambda example scenario.
* This includes instantiated aws_sdk clients and details of resource names.
*/
pub struct LambdaManager {
    iam_client: aws_sdk_iam::Client,
    lambda_client: aws_sdk_lambda::Client,
    s3_client: aws_sdk_s3::Client,
    lambda_name: String,
    role_name: String,
    bucket: String,
    own_bucket: bool
}

// These unit type structs provide nominal typing on top of String parameters for
// LambdaManager::new
pub struct LambdaName(pub String);
pub struct RoleName(pub String);
pub struct Bucket(pub String);
pub struct OwnBucket(pub bool);
impl LambdaManager {
    pub fn new(
        iam_client: aws_sdk_iam::Client,
        lambda_client: aws_sdk_lambda::Client,
        s3_client: aws_sdk_s3::Client,
        lambda_name: LambdaName,
        role_name: RoleName,
        bucket: Bucket,
        own_bucket: OwnBucket,
    ) -> Self {
        Self {
            iam_client,
            lambda_client,
            s3_client,
            lambda_name: lambda_name.0,
            role_name: role_name.0,
            bucket: bucket.0,
            own_bucket: own_bucket.0,
        }
    }
}

/**
 * Load the AWS configuration from the environment.
 * Look up lambda_name and bucket if none are given, or generate a random
 * name if not present in the environment.
 * If the bucket name is provided, the caller needs to have created the
 * bucket.
 * If the bucket name is generated, it will be created.
 */
pub async fn load_from_env(lambda_name: Option<String>, bucket: Option<String>) -> Self {
    let sdk_config = aws_config::load_from_env().await;
    let lambda_name = LambdaName(lambda_name.unwrap_or_else(|| {
        std::env::var("LAMBDA_NAME").unwrap_or_else(|_|
            "rust_lambda_example".to_string())
    }));
    let role_name = RoleName(format!("{}\_role", lambda_name.0));
    let (bucket, own_bucket) =
        match bucket {
            Some(bucket) => (Bucket(bucket), false),
            None => (Bucket(std::env::var("LAMBDA_BUCKET").unwrap_or_else(|_| {
                format!("rust-lambda-example-{}", uuid::Uuid::new_v4())
            })),
            own_bucket: own_bucket.0,
        }
}
true,
),
};

let s3_client = aws_sdk_s3::Client::new(&sdk_config);

if own_bucket {
    info!("Creating bucket for demo: {}", bucket.0);
    s3_client
        .create_bucket()
        .bucket(bucket.0.clone())
        .create_bucket_configuration(
            CreateBucketConfiguration::builder()
                .location_constraint(aws_sdk_s3::types::BucketLocationConstraint::from(
                    sdk_config.region().unwrap().as_ref(),
                ))
                .build(),
        )
        .send()
        .await
        .unwrap();
}

Self::new(
    aws_sdk_iam::Client::new(&sdk_config),
    aws_sdk_lambda::Client::new(&sdk_config),
    s3_client,
    lambda_name,
    role_name,
    bucket,
    OwnBucket(own_bucket),
)
}

// snippet-start:[lambda.rust.scenario.prepare_function]
/**
 * Upload function code from a path to a zip file.
 * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
 * The easiest way to create such a zip is to use `cargo lambda build --output-format Zip`.
 */
async fn prepare_function(
    &self,
```rust
zip_file: PathBuf,
key: Option<String>,
) -> Result<FunctionCode, anyhow::Error> {
    let body = ByteStream::from_path(zip_file).await?

    let key = key.unwrap_or_else(|| format!("{}\_code", self.lambda_name));

    info!("Uploading function code to s3://{}\{}/", self.bucket, key);
    let _ = self
        .s3_client
        .put_object()
        .bucket(self.bucket.clone())
        .key(key.clone())
        .body(body)
        .send()
        .await?

    Ok(FunctionCode::builder()
        .s3_bucket(self.bucket.clone())
        .s3_key(key)
        .build())
}
// snippet-end:[lambda.rust.scenario.prepare_function]

// snippet-start:[lambda.rust.scenario.create_function]
/**
 * Create a function, uploading from a zip file.
 */
pub async fn create_function(&self, zip_file: PathBuf) -> Result<String, anyhow::Error> {
    let code = self.prepare_function(zip_file, None).await?

    let key = code.s3_key().unwrap().to_string();

    self.create_role().await;

    let role = self
        .iam_client
        .create_role()
        .role_name(self.role_name.clone())
        .assume_role_policy_document(ROLE_POLICY_DOCUMENT)
        .send()
        .await?
```

Get started with functions
info!("Created iam role, waiting 15s for it to become active");
tokio::time::sleep(Duration::from_secs(15)).await;

info!("Creating lambda function {}", self.lambda_name);
let _ = self
    .lambda_client
    .create_function()
    .function_name(self.lambda_name.clone())
    .code(code)
    .role(role.role().map(|r| r.arn()).unwrap_or_default())
    .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
    .handler("_unused")
    .send()
    .await
    .map_err(anyhow::Error::from)?;

self.wait_for_function_ready().await?;

self.lambda_client
    .publish_version()
    .function_name(self.lambda_name.clone())
    .send()
    .await?

Ok(key)
}

async fn create_role(&self) {
    info!("Creating execution role for function");
    if let Ok(_response) = self
        .iam_client
        .get_role()
        .role_name(self.role_name.clone())
        .send()
        .await
    {
        let delete_response = self
            .iam_client
            .delete_role()
            .role_name(self.role_name.clone())

        // snippet-end:[lambda.rust.scenario.create_function]

        /**
         * Create an IAM execution role for the managed Lambda function.
         */
        async fn create_role(&self) {
            info!("Creating execution role for function");
            if let Ok(_response) = self
                .iam_client
                .get_role()
                .role_name(self.role_name.clone())
                .send()
                .await
            {
                let delete_response = self
                    .iam_client
                    .delete_role()
                    .role_name(self.role_name.clone())

        // snippet-end:[lambda.rust.scenario.create_function]
send().await;

match delete_response {
    Ok(_) => debug!("Deleted role first"),
    Err(_) => {
        warn!("Failed to delete role, will probably fail to create the new role")
    }
}

/**
* Poll `is_function_ready` with a 1-second delay. It returns when the function is ready or when there's an error checking the function's state.
*/
pub async fn wait_for_function_ready(&self) -> Result<(), anyhow::Error> {
    info!("Waiting for function");
    while !self.is_function_ready(None).await? {
        info!("Function is not ready, sleeping 1s");
        tokio::time::sleep(Duration::from_secs(1)).await;
    }
    Ok(()
}

/**
* Check if a Lambda function is ready to be invoked.
* A Lambda function is ready for this scenario when its state is active and its LastUpdateStatus is Successful.
* Additionally, if a sha256 is provided, the function must have that as its current code hash.
* Any missing properties or failed requests will be reported as an Err.
*/
async fn is_function_ready(
    &self,
    expected_code_sha256: Option<&str>,
) -> Result<bool, anyhow::Error> {
    match self.get_function().await {
        Ok(func) => {
            if let Some(config) = func.configuration() {
                if let Some(state) = config.state() {
                    info!(?state, "Checking if function is active");
                    if !matches!(state, State::Active) {
                        return Ok(false);
                    }
                }
            }
        }
    }
}
match config.last_update_status() {
    Some(last_update_status) => {
        info!(?last_update_status, "Checking if function is ready");
        match last_update_status {
            LastUpdateStatus::Successful => {
                // continue
            }
            LastUpdateStatus::Failed | LastUpdateStatus::InProgress => {
                return Ok(false);
            }
            unknown => {
                warn!(status_variant = unknown.as_str(), "LastUpdateStatus unknown");
                return Err(anyhow!("Unknown LastUpdateStatus, fn config is {config:?}"));
            }
        }
    }
    None => {
        warn!("Missing last update status");
        return Ok(false);
    }
};
if expected_code_sha256.is_none() {
    return Ok(true);
}
if let Some(code_sha256) = config.code_sha256() {
    return Ok(code_sha256 == expected_code_sha256.unwrap_or_default());
}

Err(e) => {
    warn!(?e, "Could not get function while waiting");
}
Ok(false)
}

// snippet-start:[lambda.rust.scenario.get_function]
/** Get the Lambda function with this Manager's name. */
pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error> {
    info!("Getting lambda function");
    self.lambda_client
        .get_function()
        .function_name(self.lambda_name.clone())
        .send()
        .await
        .map_err(anyhow::Error::from)
}
// snippet-end:[lambda.rust.scenario.get_function]

// snippet-start:[lambda.rust.scenario.list_functions]
/** List all Lambda functions in the current Region. */
pub async fn list_functions(&self) -> Result<ListFunctionsOutput, anyhow::Error> {
    info!("Listing lambda functions");
    self.lambda_client
        .list_functions()
        .send()
        .await
        .map_err(anyhow::Error::from)
}
// snippet-end:[lambda.rust.scenario.list_functions]

// snippet-start:[lambda.rust.scenario.invoke]
/** Invoke the lambda function using calculator InvokeArgs. */
pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput, anyhow::Error> {
    info!(?args, "Invoking ", self.lambda_name);
    let payload = serde_json::to_string(&args)?;
    debug!(?payload, "Sending payload");
    self.lambda_client
        .invoke()
        .function_name(self.lambda_name.clone())
        .payload(Blob::new(payload))
        .send()
        .await
        .map_err(anyhow::Error::from)
/** Given a Path to a zip file, update the function's code and wait for the update to finish. */
pub async fn update_function_code(
    &self,
    zip_file: PathBuf,
    key: String,
) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
    let function_code = self.prepare_function(zip_file, Some(key)).await?
    .info!("Updating code for {}", self.lambda_name);
    let update = self
        .lambda_client
        .update_function_code()
        .function_name(self.lambda_name.clone())
        .s3_bucket(self.bucket.clone())
        .s3_key(function_code.s3_key().unwrap().to_string())
        .send()
        .await
        .map_err(anyhow::Error::from)?;

    self.wait_for_function_ready().await?

    Ok(update)
}

/** Update the environment for a function. */
pub async fn update_function_configuration(
    &self,
    environment: Environment,
) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
    info!(
        "Updating environment for {}", self.lambda_name
    );
    let updated = self
        .lambda_client
        .update_function_configuration()
.environment(environment)
 .send()
 .await
 .map_err(anyhow::Error::from)?;

 self.wait_for_function_ready().await?

 Ok(updated)
}

// snippet-end:[lambda.rust.scenario.update_function_configuration]

// snippet-start:[lambda.rust.scenario.delete_function]
/** Delete a function and its role, and if possible or necessary, its
 associated code object and bucket. */
pub async fn delete_function(
 &self,
 location: Option<String>,
) -> (Result<DeleteFunctionOutput, anyhow::Error>,
 Result<DeleteRoleOutput, anyhow::Error>,
 Option<Result<DeleteObjectOutput, anyhow::Error>>,
)
{
 info!("Deleting lambda function {}", self.lambda_name);
 let delete_function = self
 .lambda_client
 .delete_function()
 .function_name(self.lambda_name.clone())
 .send()
 .await
 .map_err(anyhow::Error::from);

 info!("Deleting iam role {}", self.role_name);
 let delete_role = self
 .iam_client
 .delete_role()
 .role_name(self.role_name.clone())
 .send()
 .await
 .map_err(anyhow::Error::from);

 let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
 if let Some(location) = location {
  info!("Deleting object {location}");
  Some(}
self.s3_client
    .delete_object()
    .bucket(self.bucket.clone())
    .key(location)
    .send()
    .await
    .map_err(anyhow::Error::from),
}
}
else {
    info!(?location, "Skipping delete object");
    None
};

(delete_function, delete_role, delete_object)
}
// snippet-end:[lambda.rust.scenario.delete_function]

pub async fn cleanup(
    &self,
    location: Option<String>,
) -> (  
    (  
        Result<DeleteFunctionOutput, anyhow::Error>,
        Result<DeleteRoleOutput, anyhow::Error>,
        Option<Result<DeleteObjectOutput, anyhow::Error>>,
    ),
    Option<Result<DeleteBucketOutput, anyhow::Error>>,
) {
    let delete_function = self.delete_function(location).await;

    let delete_bucket = if self.own_bucket {
        info!("Deleting bucket ", self.bucket);
        if delete_function.2.is_none() ||
        delete_function.2.as_ref().unwrap().is_ok() {
            Some(  
                self.s3_client
                .delete_bucket()
                .bucket(self.bucket.clone())
                .send()
                .await
                .map_err(anyhow::Error::from),
            )
        } else {
            None
        }
    } else {
        None
    }
A binary to run the scenario from front to end, using command line flags to control some behavior. This file is src/bin/scenario.rs in the crate.

/*
 * Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
 * SPDX-License-Identifier: Apache-2.0.
 */
/*
## Service actions

Service actions wrap the SDK call, taking a client and any specific parameters necessary for the call.

* CreateFunction
* GetFunction
* ListFunctions
* Invoke
* UpdateFunctionCode
* UpdateFunctionConfiguration
* DeleteFunction

## Scenario

A scenario runs at a command prompt and prints output to the user on the result of each service action. A scenario can run in one of two ways: straight through, printing out progress as it goes, or as an interactive question/answer script.

## Getting started with functions

Use an SDK to manage AWS Lambda functions: create a function, invoke it, update its code, invoke it again, view its output and logs, and delete it.

This scenario uses two Lambda handlers:
_Note: Handlers don't use AWS SDK API calls._

The increment handler is straightforward:

1. It accepts a number, increments it, and returns the new value.
2. It performs simple logging of the result.

The arithmetic handler is more complex:
1. It accepts a set of actions ['plus', 'minus', 'times', 'divided-by'] and two numbers, and returns the result of the calculation.
2. It uses an environment variable to control log level (such as DEBUG, INFO, WARNING, ERROR).

It logs a few things at different levels, such as:
   * DEBUG: Full event data.
   * INFO: The calculation result.
   * WARN~ING~: When a divide by zero error occurs.
   * This will be the typical `RUST_LOG` variable.
The steps of the scenario are:

1. Create an AWS Identity and Access Management (IAM) role that meets the following requirements:
   * Has an assume_role policy that grants 'lambda.amazonaws.com' the 'sts:AssumeRole' action.
   * Attaches the 'arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole' managed role.
   * _You must wait for ~10 seconds after the role is created before you can use it!_
2. Create a function (CreateFunction) for the increment handler by packaging it as a zip and doing one of the following:
   * Adding it with CreateFunction Code.ZipFile.
   * --or--
   * Uploading it to Amazon Simple Storage Service (Amazon S3) and adding it with CreateFunction Code.S3Bucket/S3Key.
   * _Note: Zipping the file does not have to be done in code._
   * If you have a waiter, use it to wait until the function is active. Otherwise, call GetFunction until State is Active.
3. Invoke the function with a number and print the result.
4. Update the function (UpdateFunctionCode) to the arithmetic handler by packaging it as a zip and doing one of the following:
   * Adding it with UpdateFunctionCode ZipFile.
   * --or--
   * Uploading it to Amazon S3 and adding it with UpdateFunctionCode S3Bucket/S3Key.
5. Call GetFunction until Configuration.LastUpdateStatus is 'Successful' (or 'Failed').
6. Update the environment variable by calling UpdateFunctionConfiguration and pass it a log level, such as:
   * Environment={"Variables": {"RUST_LOG": 'TRACE'}}
7. Invoke the function with an action from the list and a couple of values. Include LogType='Tail' to get logs in the result. Print the result of the calculation and the log.
8. [Optional] Invoke the function to provoke a divide-by-zero error and show the log result.
9. List all functions for the account, using pagination (ListFunctions).
10. Delete the function (DeleteFunction).
11. Delete the role.

Each step should use the function created in Service Actions to abstract calling the SDK.
* /
use aws_sdk_lambda::{operation::invoke::InvokeOutput, types::Environment};
use clap::Parser;
use std::{collections::HashMap, path::PathBuf};
use tracing::{debug, info, warn};
use tracing_subscriber::EnvFilter;

use lambda_code_examples::actions::{
  InvokeArgs::{Arithmetic, Increment},
  LambdaManager, Operation,
};

#[derive(Debug, Parser)]
pub struct Opt {
  /// The AWS Region.
  #[structopt(short, long)]
  pub region: Option<String>,

  // The bucket to use for the FunctionCode.
  #[structopt(short, long)]
  pub bucket: Option<String>,

  // The name of the Lambda function.
  #[structopt(short, long)]
  pub lambda_name: Option<String>,

  // The number to increment.
  #[structopt(short, long, default_value = "12")]
  pub inc: i32,

  // The left operand.
  #[structopt(long, default_value = "19")]  
  pub num_a: i32,

  // The right operand.
  #[structopt(long, default_value = "23")]  
  pub num_b: i32,

  // The arithmetic operation.
  #[structopt(short, long, default_value = "plus")]  
  pub operation: Operation,

  #[structopt(long)]
  pub cleanup: Option<bool>,
}
fn code_path(lambda: &str) -> PathBuf {
    PathBuf::from(format!("../target/lambda/{}\bootstrap.zip"))
}

// snippet-start:[lambda.rust.scenario.log_invoke_output]
fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
        info!("Could not extract payload")
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
        debug!("Invoked function had no logs")
    }
}
// snippet-end:[lambda.rust.scenario.log_invoke_output]

async fn main_block(
    opt: &Opt,
    manager: &LambdaManager,
    code_location: String,
) -> Result<(), anyhow::Error> {
    let invoke = manager.invoke(Increment(opt.inc)).await?
        .log_invoke_output(&invoke, "Invoked function configured as increment");

    let update_code = manager
        .update_function_code(code_path("arithmetic"), code_location.clone())
        .await?
    let code_sha256 = update_code.code_sha256().unwrap_or("Unknown SHA")
        .log_invoke_output(&invoke, "Updated function code with arithmetic.zip");

    let arithmetic_args = Arithmetic(opt.operation, opt.num_a, opt.num_b);
    let invoke = manager.invoke(arithmetic_args).await?
        .log_invoke_output(&invoke, "Invoked function configured as arithmetic");

    let update = manager
    .update_function_code(code_path("arithmetic"), code_location.clone())
    .await?
    .log_invoke_output(&invoke, "Updated function code with arithmetic.zip");
let invoke = manager
    .invoke(Arithmetic(opt.operation, opt.num_a, opt.num_b))
    .await?
log_invoke_output(
    &invoke,
    "Invoked function configured as arithmetic with increased logging",
);

let invoke = manager
    .invoke(Arithmetic(Operation::DividedBy, opt.num_a, 0))
    .await?
log_invoke_output(
    &invoke,
    "Invoked function configured as arithmetic with divide by zero",
);

Ok::<(), anyhow::Error>(()
}
let key = match manager.create_function(code_path("increment")).await {
    Ok(init) => {
        info!(?init, "Created function, initially with increment.zip");
        let run_block = main_block(&opt, &manager, init.clone()).await;
        info!(?run_block, "Finished running example, cleaning up");
        Some(init)
    }
    Err(err) => {
        warn!(?err, "Error happened when initializing function");
        None
    }
};

if Some(false) == opt.cleanup || Some(true) == opt.no_cleanup {
    info!("Skipping cleanup")
} else {
    let delete = manager.cleanup(key).await;
    info!(?delete, "Deleted function & cleaned up resources");
}
TRY.

"Create an AWS Identity and Access Management (IAM) role that grants AWS Lambda permission to write to logs."

DATA(lv_policy_document) = `{` &&
` "Version": "2012-10-17",` &&
` "Statement": [` &&
` {` &&
` "Effect": "Allow",` &&
` "Action": [` &&
` "sts:AssumeRole"` &&
` ],` &&
` "Principal": {` &&
` "Service": [` &&
` "lambda.amazonaws.com"` &&
` ],` &&
` `{` &&
` `}]` &&
` `].` &&
` `
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}
` `}`.

TRY.

DATA(lo_create_role_output) = lo_iam->createrole(
  iv_rolename = iv_role_name
  iv_assumerolepolicydocument = lv_policy_document
  iv_description = 'Grant lambda permission to write to logs'
).

MESSAGE 'IAM role created.' TYPE 'I'.

WAIT UP TO 10 SECONDS. " Make sure that the IAM role is ready for use. "

CATCH /aws1/cx_iamentityalrdyexex.

MESSAGE 'IAM role already exists.' TYPE 'E'.

CATCH /aws1/cx_iaminvalidinputex.
TRY.
    lo_iam->attachrolememberpolicy(
        iv_rolemember = iv_role_name
        iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
        AWSLambdaBasicExecutionRole'
    )
    MESSAGE 'Attached policy to the IAM role.' TYPE 'I'.
CATCH /aws1/cx_iaminvalidinputex.
    MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
CATCH /aws1/cx_iamnosuchentityex.
    MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
CATCH /aws1/cx_iamplynotattachableex.
    MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
CATCH /aws1/cx_iamunmodableentityex.
    MESSAGE 'Service that depends on the service-linked role is not
modifiable.' TYPE 'E'.
ENDTRY.

" Create a Lambda function and upload handler code. "
" Lambda function performs 'increment' action on a number. "
TRY.
    lo_lmd->createfunction(
        iv_functionname = iv_function_name
        iv_runtime = `python3.9`
        iv_role = lo_create_role_output->get_role( )->get_arn( )
        iv_handler = iv_handler
        io_code = io_initial_zip_file
        iv_description = 'AWS Lambda code example'
    )
    MESSAGE 'Lambda function created.' TYPE 'I'.
CATCH /aws1/cx_lmdcodestorageexcdex.
    MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
CATCH /aws1/cx_lmdinvparamvalueex.
    MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
CATCH /aws1/cx_lmdresourcenotfoundex.
    MESSAGE 'The requested resource does not exist.' TYPE 'E'.
ENDTRY.
"Verify the function is in Active state"
WHILE lo_lmd->getfunction( iv_functionname = iv_function_name )-
   >get_configuration( )->ask_state( ) <> 'Active'.
   IF sy-index = 10.
      EXIT.               " Maximum 10 seconds. "
   ENDIF.
   WAIT UP TO 1 SECONDS.
ENDWHILE.

"Invoke the function with a single parameter and get results."
TRY.
   DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring( 
      '{
           "action": "increment", 
           "number": 10
      }' ).
   DATA(lo_initial_invoke_output) = lo_lmd->invoke( 
      iv_functionname = iv_function_name 
      iv_payload = lv_json
   ).
   ov_initial_invoke_payload = lo_initial_invoke_output->get_payload( ).
   " ov_initial_invoke_payload is returned for testing purposes. "
   DATA(lo_writer_json) = cl_sxml_string_writer=>create( type = 
      if_sxml=>co_xt_json ).
   CALL TRANSFORMATION id SOURCE XML ov_initial_invoke_payload RESULT 
      XML lo_writer_json.
   DATA(lv_result) = cl_abap_codepage=>convert_from( lo_writer_json-
      >get_output( ) ).
      MESSAGE 'Lambda function invoked.' TYPE 'I'.
   CATCH /aws1/cx_lmdinvparamvalueex.
      MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
   CATCH /aws1/cx_lmdinvrequestcontext.
      MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
   CATCH /aws1/cx_lmdresourcenotfoundex.
      MESSAGE 'The requested resource does not exist.' TYPE 'E'.
   CATCH /aws1/cx_lmdunsuppedmediatyp00.
      MESSAGE 'Invoke request body does not have JSON as its content type.'
      TYPE 'E'.
   ENDTRY.

" Update the function code and configure its Lambda environment with an
environment variable. "
" Lambda function is updated to perform 'decrement' action also. "
TRY.
lo_lmd->updatefunctioncode(
    iv_functionname = iv_function_name
    iv_zipfile = io_updated_zip_file
).
WAIT UP TO 10 SECONDS. " Make sure that the update is completed."
MESSAGE 'Lambda function code updated.' TYPE 'I'.
CATCH /aws1/cx_lmdcodestorageexcdex.
    MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
CATCH /aws1/cx_lmdinvparamvalueex.
    MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
CATCH /aws1/cx_lmdresourcenotfoundex.
    MESSAGE 'The requested resource does not exist.' TYPE 'E'.
ENDTRY.

TRY.
DATA lt_variables TYPE /aws1/ cl_lmdenvironmentvaria00=>tt_environmentvariables.
    DATA ls_variable LIKE LINE OF lt_variables.
    ls_variable-key = 'LOG_LEVEL'.
    ls_variable-value = NEW /aws1/cl_lmdenvironmentvaria00( iv_value = 'info' ).
    INSERT ls_variable INTO TABLE lt_variables.
lo_lmd->updatefunctionconfiguration(
    iv_functionname = iv_function_name
    io_environment = NEW /aws1/cl_lmdenvironment( it_variables = lt_variables )
).
WAIT UP TO 10 SECONDS. " Make sure that the update is completed."
MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
CATCH /aws1/cx_lmdinvparamvalueex.
    MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
CATCH /aws1/cx_lmdresourceconflictex.
    MESSAGE 'Resource already exists or another operation is in progress.' TYPE 'E'.
CATCH /aws1/cx_lmdresourcenotfoundex.
    MESSAGE 'The requested resource does not exist.' TYPE 'E'.
ENDTRY.

"Invoke the function with new parameters and get results. Display the execution log that's returned from the invocation."
TRY.
   lv_json = /aws1/cl_rt_util=>string_to_xstring(
      '{
         "action": "decrement", 
         "number": 10
      }'
   ).
   DATA(lo_updated_invoke_output) = lo_lmd->invoke(
      iv_functionname = iv_function_name
      iv_payload = lv_json
   ).
   ov_updated_invoke_payload = lo_updated_invoke_output->get_payload( ).
   " ov_updated_invoke_payload is returned for testing purposes. "
   lo_writer_json = cl_sxml_string_writer=>create( type =
      if_sxml=>co_xt_json )
   CALL TRANSFORMATION id SOURCE XML ov_updated_invoke_payload RESULT XML lo_writer_json.
   lv_result = cl_abap_codepage=>convert_from( lo_writer_json->get_output( ) ).
   MESSAGE 'Lambda function invoked.' TYPE 'I'.
   CATCH /aws1/cx_lmdinvparamvalueex.
      MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
   CATCH /aws1/cx_lmdinvrequestcontext.
      MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
   CATCH /aws1/cx_lmdresourcenooutput.
      MESSAGE 'The requested resource does not exist.' TYPE 'E'.
   CATCH /aws1/cx_lmdunsupmediatyp00.
      MESSAGE 'Invoke request body does not have JSON as its content type.' TYPE 'E'.
   ENDTRY.

" List the functions for your account. "
TRY.
   DATA(lo_list_output) = lo_lmd->listfunctions( ).
   DATA(lt_functions) = lo_list_output->get_functions( ).
   MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
   CATCH /aws1/cx_lmdinvparamvalueex.
      MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
   ENDTRY.

" Delete the Lambda function. "
TRY.
   lo_lmd->deletefunction( iv_functionname = iv_function_name ).
   MESSAGE 'Lambda function deleted.' TYPE 'I'.

CATCH /aws1/cx_lmdinvparamvalueex.
    MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
CATCH /aws1/cx_lmdresourcenotfoundex.
    MESSAGE 'The requested resource does not exist.' TYPE 'E'.
ENDTRY.

" Detach role policy. "
TRY.
    lo_iam->detachrolepolicy(
        iv_rolename = iv_role_name
        iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole'
    ).
    MESSAGE 'Detached policy from the IAM role.' TYPE 'I'.
CATCH /aws1/cx_iaminvalidinputex.
    MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
CATCH /aws1/cx_iamnosuchentityex.
    MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
CATCH /aws1/cx_iamplynotattachableex.
    MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
CATCH /aws1/cx_iamunmodableentityex.
    MESSAGE 'Service that depends on the service-linked role is not
modifiable.' TYPE 'E'.
ENDTRY.

" Delete the IAM role. "
TRY.
    lo_iam->deleterole( iv_rolename = iv_role_name ).
    MESSAGE 'IAM role deleted.' TYPE 'I'.
CATCH /aws1/cx_iamnosuchentityex.
    MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
CATCH /aws1/cx_iamunmodableentityex.
    MESSAGE 'Service that depends on the service-linked role is not
modifiable.' TYPE 'E'.
ENDTRY.

CATCH /aws1/cx_rt_service_generic INTO lo_exception.
    DATA(lv_error) = lo_exception->get_longtext( ).
    MESSAGE lv_error TYPE 'E'.
ENDTRY.

• For API details, see the following topics in AWS SDK for SAP ABAP API reference.
For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

**Serverless examples for Lambda using AWS SDKs**

The following code examples show how to use Lambda with AWS SDKs.

**Examples**

- **Invoke a Lambda function from a Kinesis trigger**
- **Invoke a Lambda function from an Amazon S3 trigger**
- **Invoke a Lambda function from an Amazon SNS trigger**
- **Invoke a Lambda function from an Amazon SQS trigger**
- **Reporting batch item failures for Lambda functions with a Kinesis trigger**
- **Reporting batch item failures for Lambda functions with an Amazon SQS trigger**

**Invoke a Lambda function from a Kinesis trigger**

The following code examples show how to implement a Lambda function that receives an event triggered by receiving records from a Kinesis stream. The function retrieves the Kinesis payload, decodes from Base64, and logs the record contents.
### AWS SDK for .NET

#### Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

---

**Consuming a Kinesis event with Lambda using .NET.**

```csharp
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
[assembly:
LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer))]

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId: {record.EventId}"");
            }
        }
    }
}
```
Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using Go.

```go
package main

import {
    "context"
    "log"
}

string data = await GetRecordDataAsync(record.Kinesis, context);
Logger.LogInformation("Data: {data}");
// TODO: Do interesting work based on the new data
}
catch (Exception ex)
{
    Logger.LogError("An error occurred {ex.Message}");
    throw;
}
}
Logger.LogInformation("Successfully processed {evnt.Records.Count} records.");

private async Task<string> GetRecordDataAsync(KinesisEvent.Record record, ILambdaContext context)
{
    byte[] bytes = record.Data.ToArray();
    string data = Encoding.UTF8.GetString(bytes);
    await Task.CompletedTask; //Placeholder for actual async work
    return data;
}
```
func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
    if len(kinesisEvent.Records) == 0 {
        log.Printf("empty Kinesis event received")
        return nil
    }

    for _, record := range kinesisEvent.Records {
        log.Printf("processed Kinesis event with EventId: %v", record.EventID)
        recordDataBytes := record.Kinesis.Data
        recordDataText := string(recordDataBytes)
        log.Printf("record data: %v", recordDataText)
        // TODO: Do interesting work based on the new data
    }

    log.Printf("successfully processed %v records", len(kinesisEvent.Records))
    return nil
}

func main() {
    lambda.Start(handler)
}

### Note
There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](https://github.com/aws-samples/serverless-go-adapter) repository.

Consuming a Kinesis event with Lambda using Java.

```java
package example;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
```
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+" records");
        return null;
    }
}

JavaScript

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using JavaScript.
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}

Consuming a Kinesis event with Lambda using TypeScript.

import { KinesisStreamEvent,
         Context,
         KinesisStreamHandler,
         KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
    logLevel: "INFO",
    serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (event: KinesisStreamEvent,
                                                                 context: Context
): Promise<void> => {
  for (const record of event.Records) {

try {
    const recordData = await getRecordDataAsync(record.kinesis);
    // TODO: Do interesting work based on the new data
} catch (err) {
    throw err;
}

async function getRecordDataAsync(
    payload: KinesisStreamRecordPayload
): Promise<string> {
    var data = Buffer.from(payload.data, "base64").toString("utf-8");
    return data;
}

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using Python.

```python
import base64

def lambda_handler(event, context):
    for record in event['Records']:
        try:
            record_data = base64.b64decode(record['kinesis']['data']).decode('utf-8')
```
For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK](https://docs.aws.amazon.com/sdk-for-cpp/v1/developer-guide/html). This topic also includes information about getting started and details about previous SDK versions.

## Invoke a Lambda function from an Amazon S3 trigger

The following code examples show how to implement a Lambda function that receives an event triggered by uploading an object to an S3 bucket. The function retrieves the S3 bucket name and object key from the event parameter and calls the Amazon S3 API to retrieve and log the content type of the object.

### .NET

#### AWS SDK for .NET

```csharp
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
```

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](https://github.com/aws-samples/serverless-aws-lambda) repository.
namespace S3Integration
{
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }

        public async Task<string> Handler(S3Event evt, ILambdaContext context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
                var key = HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");
                var objectResult = await _s3Client.GetObjectAsync(bucket, key);

                context.Logger.LogLine($"Returning {objectResult.Key}");
                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request - {e.Message}");
            }

            return string.Empty;
        }
    }
}
Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Go.

```go
package main

import (
    "context"
    "log"
    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/s3"
)

func handler(ctx context.Context, s3Event events.S3Event) error {
    sdkConfig, err := config.LoadDefaultConfig(ctx)
    if err != nil {
        log.Printf("failed to load default config: %s", err)
        return err
    }
    s3Client := s3.NewFromConfig(sdkConfig)

    for _, record := range s3Event.Records {
        key := record.S3.Object.URLDecodedKey
        headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
            Bucket: &bucket,
            Key: key,
        })
        if err != nil {
            log.Printf("failed to get head object: %s", err)
            return err
        }
        // Process the object
    }
}
```
Key:  &key,
})
if err != nil {
    log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
    return err
}
log.Printf("successfully retrieved %s/%s of type %s", bucket, key, *headOutput.ContentType)
}
return nil
}

func main() {
    lambda.Start(handler)
}

Java

SDK for Java 2.x

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Java.

package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNotificationRecord;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger = LoggerFactory.getLogger(Handler.class);

    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
            S3EventNotificationRecord record = s3event.getRecords().get(0);
            String srcBucket = record.getS3().getBucket().getName();
            String srcKey = record.getS3().getObject().getUrlDecodedKey();

            S3Client s3Client = S3Client.builder().build();
            HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket, srcKey);

            logger.info("Successfully retrieved "+ srcBucket + "/" + srcKey + " of type " + headObject.contentType());

            return "Ok";
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String bucket, String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
Consuming an S3 event with Lambda using JavaScript.

```javascript
const aws = require('aws-sdk');

const s3 = new aws.S3({ apiVersion: '2006-03-01' });

exports.handler = async (event, context) => {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, ' '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.headObject(params).promise();
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make sure they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

Consuming an S3 event with Lambda using TypeScript.

```typescript
import { S3Event } from 'aws-lambda';
```
```javascript
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> => {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, ' '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make sure they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

**Python**

**SDK for Python (Boto3)**

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

**Consuming an S3 event with Lambda using Python.**

```python
import json
import urllib.parse
import boto3
```
print('Loading function')

s3 = boto3.client('s3')

def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'],
                                     encoding='utf-8')

    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they exist and
        your bucket is in the same region as this function.'.format(key, bucket))
        raise e

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run
in the Serverless examples repository.

Consuming an S3 event with Lambda using Rust.

```rust
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Main function
#[tokio::main]
```
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
    tracing::info!(records = ?evt.payload.records.len(), "Received request from SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket = evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket name to exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object key to exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;

    match s3_get_object_result {

    }
Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult contains a 'body' property of type ByteStream"),
    Err(_) => tracing::info!("Failure with S3 Get Object request")
}

Ok(())
}

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

**Invoke a Lambda function from an Amazon SNS trigger**

The following code examples show how to implement a Lambda function that receives an event triggered by receiving messages from an SNS topic. The function retrieves the messages from the event parameter and logs the content of each message.

**.NET**

**AWS SDK for .NET**

> Note
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using .NET.

```csharp
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
[assembly:
    LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer))]

namespace SnsIntegration;
```
public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record, ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation("Processed record
            {record.Sns.Message}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            // You can use Dead Letter Queue to handle failures. By configuring a Lambda DLQ.
            context.Logger.LogError("An error occurred");
            throw;
        }
    }
}

Go

SDK for Go V2

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.
Consuming an SNS event with Lambda using Go.

```go
docker
package main

import (
    "context"
    "fmt"

    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
    for _, record := range snsEvent.Records {
        processMessage(record)
    }
    fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
    message := record.SNS.Message
    fmt.Printf("Processed message: %s\n", message)
    // TODO: Process your record here
}

func main() {
    lambda.Start(handler)
}
```

JavaScript

**SDK for JavaScript (v2)**

ℹ️ Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming an SNS event with Lambda using JavaScript.
exports.handler = async (event, context) => {
    for (const record of event.Records) {
        await processMessageAsync(record);
    }
    console.info("done");
};

async function processMessageAsync(record) {
    try {
        const message = JSON.stringify(record.Sns.Message);
        console.log(`Processed message ${message}`);
        await Promise.resolve(1); //Placeholder for actual async work
    } catch (err) {
        console.error("An error occurred");
        throw err;
    }
}

Consuming an SNS event with Lambda using TypeScript.

import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (event: SNSEvent, context: Context): Promise<void> => {
    for (const record of event.Records) {
        await processMessageAsync(record);
    }
    console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
    try {
        const message: string = JSON.stringify(record.Sns.Message);
        console.log("Processed message ${message}");
        await Promise.resolve(1); //Placeholder for actual async work
    } catch (err) {
        console.error("An error occurred");
        throw err;
    }
}
Consuming an SNS event with Lambda using Python.

```python
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}")
        # TODO; Process your record here
    except Exception as e:
        print("An error occurred")
        raise e
```

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK](https://aws.amazon.com/sdk-for-python/). This topic also includes information about getting started and details about previous SDK versions.

### Invoke a Lambda function from an Amazon SQS trigger

The following code examples show how to implement a Lambda function that receives an event triggered by receiving messages from an SQS queue. The function retrieves the messages from the event parameter and logs the content of each message.
Consuming an SQS event with Lambda using .NET.

using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
// into a .NET class.

namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message, ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation("Processed message {message.Body}";
            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
    }
}
catch (Exception e)
{
    // You can use Dead Letter Queue to handle failures. By configuring a Lambda DLQ.
    context.Logger.LogError("An error occurred");
    throw;
}

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using Go.

package integration_sqs_to_lambda

import (
    "fmt"
    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
    for _, record := range event.Records {
        err := processMessage(record)
        if err != nil {
            return err
        }
    }
    fmt.Println("done")
    return nil
}
func processMessage(record events.SQSMessage) error {
    fmt.Printf("Processed message %s\n", record.Body)
    // TODO: Do interesting work based on the new message
    return nil
}

func main() {
    lambda.Start(handler)
}

Java

SDK for Java 2.x

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
        return null;
    }

    private void processMessage(SQSEvent msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());
        } catch (Exception e) {
            context.getLogger().log("Error processing message: ": e.getMessage());
        }
    }
}

Note

There's more on GitHub. Find the complete example and learn how to set up and run
in the Serverless examples repository.

Consuming an SQS event with Lambda using Java.
JavaScript

SDK for JavaScript (v2)

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using JavaScript.

```javascript
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```
Consuming an SQS event with Lambda using TypeScript.

```typescript
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";

export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Python

**SDK for Python (Boto3)**

```python
def lambda_handler(event, context):
  for message in event['Records']:
    process_message(message)
```

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](https://github.com/awslabs/serverless-example) repository.
print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}"")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err

Ruby

**SDK for Ruby**

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](mailto:Serverless examples) repository.

Consuming an SQS event with Lambda using Ruby.

def lambda_handler(event:, context:)
    event['Records'].each do |message|
        process_message(message)
    end
    puts "done"
end

def process_message(message)
    begin
        puts "Processed message #{message['body']}"
        # TODO: Do interesting work based on the new message
    rescue StandardError => err
        puts "An error occurred"
        raise err
    end
end
For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

**Reporting batch item failures for Lambda functions with a Kinesis trigger**

The following code examples show how to implement partial batch response for Lambda functions that receive events from a Kinesis stream. The function reports the batch item failures in the response, signaling to Lambda to retry those messages later.

**.NET**

**AWS SDK for .NET**

---

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

---

Reporting Kinesis batch item failures with Lambda using .NET.

```csharp
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
[assembly:
]

namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]

```
public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt, ILambdaContext context)
{
    if (evnt.Records.Count == 0)
    {
        Logger.LogInformation("Empty Kinesis Event received");
        return new StreamsEventResponse();
    }

    foreach (var record in evnt.Records)
    {
        try
        {
            Logger.LogInformation("Processed Event with EventId: {record.EventId} ");
            string data = await GetRecordDataAsync(record.Kinesis, context);
            Logger.LogInformation("Data: {data}");
            // TODO: Do interesting work based on the new data
        }
        catch (Exception ex)
        {
            Logger.LogError("An error occurred {ex.Message}";)
            /* Since we are working with streams, we can return the failed item immediately.
               Lambda will immediately begin to retry processing from this failed item onwards. */
            return new StreamsEventResponse
            {
                BatchItemFailures = new
                List<StreamsEventResponse.BatchItemFailure>
                {
                    new StreamsEventResponse.BatchItemFailure
                    {
                        ItemIdentifier = record.Kinesis.SequenceNumber
                    }
                }
            }
        }
    }

    Logger.LogInformation("Successfully processed {evnt.Records.Count} records.");
    return new StreamsEventResponse();
}

private async Task<string> GetRecordDataAsync(KinesisEvent.Record record, ILambdaContext context)
public class StreamsEventResponse {
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
StreamsEventResponse> {

    Reporting batch item failures for Lambda functions with a Kinesis trigger
    1984

    Java

    SDK for Java 2.x

    Note
    There's more on GitHub. Find the complete example and learn how to set up and run
    in the Serverless examples repository.
@Override
public StreamsEventResponse handleRequest(KinesisEvent input, Context context) {
    List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new ArrayList<>();
    String curRecordSequenceNumber = "";

    for (KinesisEvent.KinesisEventRecord kinesisEventRecord : input.getRecords()) {
        try {
            //Process your record
            KinesisEvent.Record kinesisRecord = kinesisEventRecord.getKinesis();
            curRecordSequenceNumber = kinesisRecord.getSequenceNumber();
        }
        catch (Exception e) {
            /* Since we are working with streams, we can return the failed item immediately.
               Lambda will immediately begin to retry processing from this failed item onwards. */
            batchItemFailures.add(new StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
            return new StreamsEventResponse(batchItemFailures);
        }
    }

    return new StreamsEventResponse(batchItemFailures);
}

JavaScript

SDK for JavaScript (v2)

Note
There’s more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.
Reporting Kinesis batch item failures with Lambda using Javascript.

```javascript
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
      immediately.
      Lambda will immediately begin to retry processing from this failed
      item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Reporting Kinesis batch item failures with Lambda using TypeScript.

```typescript
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";
```
```javascript
const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (event: KinesisStreamEvent, context: Context): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item immediately. 
      Lambda will immediately begin to retry processing from this failed item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      }
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload: KinesisStreamRecordPayload): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

**Reporting batch item failures for Lambda functions with a Kinesis trigger**
 Reporting Kinesis batch item failures with Lambda using Python.

```python
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {
                "batchItemFailures": [
                    {
                        "itemIdentifier": curRecordSequenceNumber
                    }
                ]
            }
        return {
            "batchItemFailures": []
        }
```

Reporting batch item failures for Lambda functions with an Amazon SQS trigger

The following code examples show how to implement partial batch response for Lambda functions that receive events from an SQS queue. The function reports the batch item failures in the response, signaling to Lambda to retry those messages later.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.
Reporting SQS batch item failures with Lambda using .NET.

using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
[assembly:
LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer))]
namespace sqsSample;
public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
            {
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
            {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.Add(new SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }
}
private async Task ProcessMessageAsync(SQSEvent.SQSMessage message, ILambdaContext context)
{
    if (String.IsNullOrEmpty(message.Body))
    {
        throw new Exception("No Body in SQS Message.");
    }
    context.Logger.LogInformation("Processed message {message.Body}");
    // TODO: Do interesting work based on the new message
    await Task.CompletedTask;
}

Java

SDK for Java 2.x

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using Java.

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent, SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context) {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new ArrayList<SQSBatchResponse.BatchItemFailure>();
        // ...
```java
String messageId = "";
for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
    try {
        //process your message
        messageId = message.getMessageId();
    } catch (Exception e) {
        //Add failed message identifier to the batchItemFailures list
        batchItemFailures.add(new SQSBatchResponse.BatchItemFailure(messageId));
    }

return new SQSBatchResponse(batchItemFailures);
}
```

JavaScript

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using JavaScript.

```javascript
exports.handler = async (event, context) => {
    const batchItemFailures = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ ItemIdentifier: record.messageId });
        }
    }

    return {
        statusCode: 200,
        body: JSON.stringify({ batchItemFailures })
    }
```
async function processMessageAsync(record, context) {
    if (!record.body) {
        throw new Error('No Body in SQS Message.');
    }
    context.log(`Processed message ${record.body}`);
}

Python

SDK for Python (Boto3)

🔍 Note
There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Reporting SQS batch item failures with Lambda using Python.

```python
import json
def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}

        for record in event['Records']:
            try:
                # process message
            except Exception as e:
                batch_item_failures.append({'itemIdentifier': record['messageId']})

        sqs_batch_response['batchItemFailures'] = batch_item_failures
        return sqs_batch_response
```
AWS Lambda

Developer Guide

Ruby
SDK for Ruby

Note
There's more on GitHub. Find the complete example and learn how to set up and run
in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using Ruby.
require 'json'
def lambda_handler(event:, context:)
if event
batch_item_failures = []
sqs_batch_response = {}
event["Records"].each do |record|
begin
# process message
rescue StandardError => e
batch_item_failures << {"itemIdentifier" => record['messageId']}
end
end
sqs_batch_response["batchItemFailures"] = batch_item_failures
return sqs_batch_response
end
end

Reporting batch item failures for Lambda functions with an Amazon SQS trigger

1993


Rust

SDK for Rust

Use AWS Lambda with the Rust SDK.

```rust
use aws_lambda_events::*;
    event::sqs::{SqsBatchResponse, SqsEvent},
sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}

async fn function_handler(event: LambdaEvent<SqsEvent>) ->
Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }
    Ok(SqsBatchResponse {
        batch_item_failures,
    })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](https://github.com/aws/serverless-event-driven-react-app) repository.
For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Cross-service examples for Lambda using AWS SDKs

The following sample applications use AWS SDKs to combine Lambda with other AWS services. Each example includes a link to GitHub, where you can find instructions on how to set up and run the application.

Examples

- Create an API Gateway REST API to track COVID-19 data
- Create a lending library REST API
- Create a messenger application with Step Functions
- Create a photo asset management application that lets users manage photos using labels
- Create a websocket chat application with API Gateway
- Create an application that analyzes customer feedback and synthesizes audio
- Invoke a Lambda function from a browser
- Use API Gateway to invoke a Lambda function
- Use Step Functions to invoke Lambda functions
- Use scheduled events to invoke a Lambda function

Create an API Gateway REST API to track COVID-19 data

The following code example shows how to create a REST API that simulates a system to track daily cases of COVID-19 in the United States, using fictional data.

Python

SDK for Python (Boto3)

Shows how to use AWS Chalice with the AWS SDK for Python (Boto3) to create a serverless REST API that uses Amazon API Gateway, AWS Lambda, and Amazon DynamoDB. The REST
API simulates a system that tracks daily cases of COVID-19 in the United States, using fictional data. Learn how to:

- Use AWS Chalice to define routes in Lambda functions that are called to handle REST requests that come through API Gateway.
- Use Lambda functions to retrieve and store data in a DynamoDB table to serve REST requests.
- Define table structure and security role resources in an AWS CloudFormation template.
- Use AWS Chalice and CloudFormation to package and deploy all necessary resources.
- Use CloudFormation to clean up all created resources.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](https://github.com/).

**Services used in this example**
- API Gateway
- AWS CloudFormation
- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK](https://docs.aws.amazon.com/sdk-for-python/latest/). This topic also includes information about getting started and details about previous SDK versions.

**Create a lending library REST API**

The following code example shows how to create a lending library where patrons can borrow and return books by using a REST API backed by an Amazon Aurora database.

**Python**

**SDK for Python (Boto3)**

Shows how to use the AWS SDK for Python (Boto3) with the Amazon Relational Database Service (Amazon RDS) API and AWS Chalice to create a REST API backed by an Amazon Aurora database. The web service is fully serverless and represents a simple lending library where patrons can borrow and return books. Learn how to:

- Create and manage a serverless Aurora database cluster.
• Use AWS Secrets Manager to manage database credentials.
• Implement a data storage layer that uses Amazon RDS to move data into and out of the database.
• Use AWS Chalice to deploy a serverless REST API to Amazon API Gateway and AWS Lambda.
• Use the Requests package to send requests to the web service.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example
• API Gateway
• Aurora
• Lambda
• Secrets Manager

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create a messenger application with Step Functions

The following code example shows how to create an AWS Step Functions messenger application that retrieves message records from a database table.

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) with AWS Step Functions to create a messenger application that retrieves message records from an Amazon DynamoDB table and sends them with Amazon Simple Queue Service (Amazon SQS). The state machine integrates with an AWS Lambda function to scan the database for unsent messages.

• Create a state machine that retrieves and updates message records from an Amazon DynamoDB table.
• Update the state machine definition to also send messages to Amazon Simple Queue Service (Amazon SQS).
• Start and stop state machine runs.
• Connect to Lambda, DynamoDB, and Amazon SQS from a state machine by using service integrations.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

**Services used in this example**

• DynamoDB
• Lambda
• Amazon SQS
• Step Functions

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

**Create a photo asset management application that lets users manage photos using labels**

The following code examples show how to create a serverless application that lets users manage photos using labels.

**.NET**

**AWS SDK for .NET**

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

**Services used in this example**

• API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

C++

SDK for C++

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Java

SDK for Java 2.x

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.
Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Kotlin

SDK for Kotlin

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.
For a deep dive into the origin of this example see the post on AWS Community.

**Services used in this example**

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

**PHP**

**SDK for PHP**

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

**Services used in this example**

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

**Rust**

**SDK for Rust**

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.
For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create a websocket chat application with API Gateway

The following code example shows how to create a chat application that is served by a websocket API built on Amazon API Gateway.

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) with Amazon API Gateway V2 to create a websocket API that integrates with AWS Lambda and Amazon DynamoDB.

- Create a websocket API served by API Gateway.
- Define a Lambda handler that stores connections in DynamoDB and posts messages to other chat participants.
- Connect to the websocket chat application and send messages with the Websockets package.

For complete source code and instructions on how to set up and run, see the full example on GitHub.
Services used in this example

- API Gateway
- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create an application that analyzes customer feedback and synthesizes audio

The following code examples show how to create an application that analyzes customer comment cards, translates them from their original language, determines their sentiment, and generates an audio file from the translated text.

.NET

AWS SDK for .NET

This example application analyzes and stores customer feedback cards. Specifically, it fulfills the need of a fictitious hotel in New York City. The hotel receives feedback from guests in various languages in the form of physical comment cards. That feedback is uploaded into the app through a web client. After an image of a comment card is uploaded, the following steps occur:

- Text is extracted from the image using Amazon Textract.
- Amazon Comprehend determines the sentiment of the extracted text and its language.
- The extracted text is translated to English using Amazon Translate.
- Amazon Polly synthesizes an audio file from the extracted text.

The full app can be deployed with the AWS CDK. For source code and deployment instructions, see the project in GitHub.

Services used in this example

- Amazon Comprehend
- Lambda
Java

**SDK for Java 2.x**

This example application analyzes and stores customer feedback cards. Specifically, it fulfills the need of a fictitious hotel in New York City. The hotel receives feedback from guests in various languages in the form of physical comment cards. That feedback is uploaded into the app through a web client. After an image of a comment card is uploaded, the following steps occur:

- Text is extracted from the image using Amazon Textract.
- Amazon Comprehend determines the sentiment of the extracted text and its language.
- The extracted text is translated to English using Amazon Translate.
- Amazon Polly synthesizes an audio file from the extracted text.

The full app can be deployed with the AWS CDK. For source code and deployment instructions, see the project in [GitHub](https://github.com).

**Services used in this example**

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

JavaScript

**SDK for JavaScript (v3)**

This example application analyzes and stores customer feedback cards. Specifically, it fulfills the need of a fictitious hotel in New York City. The hotel receives feedback from guests in various languages in the form of physical comment cards. That feedback is uploaded into
the app through a web client. After an image of a comment card is uploaded, the following steps occur:

- Text is extracted from the image using Amazon Textract.
- Amazon Comprehend determines the sentiment of the extracted text and its language.
- The extracted text is translated to English using Amazon Translate.
- Amazon Polly synthesizes an audio file from the extracted text.

The full app can be deployed with the AWS CDK. For source code and deployment instructions, see the project in GitHub. The following excerpts show how the AWS SDK for JavaScript is used inside of Lambda functions.

```javascript
import {
    ComprehendClient,
    DetectDominantLanguageCommand,
    DetectSentimentCommand,
} from "@aws-sdk/client-comprehend";

/**
 * Determine the language and sentiment of the extracted text.
 *
 * @param {{ source_text: string}} extractTextOutput
 */
export const handler = async (extractTextOutput) => {
    const comprehendClient = new ComprehendClient({});

    const detectDominantLanguageCommand = new DetectDominantLanguageCommand(
        { Text: extractTextOutput.source_text, }
    );

    // The source language is required for sentiment analysis and
    // translation in the next step.
    const { Languages } = await comprehendClient.send(
        detectDominantLanguageCommand, 
    );

    const languageCode = Languages[0].LanguageCode;

    const detectSentimentCommand = new DetectSentimentCommand(
        { Text: extractTextOutput.source_text, LanguageCode: languageCode, }
    );

    // The source language is required for sentiment analysis and
    // translation in the next step.
    const { Text: source_text, LanguageCode: languageCode, } = extractTextOutput;

    const comprehendClient = new ComprehendClient({});

    const detectDominantLanguageCommand = new DetectDominantLanguageCommand(
        { Text: source_text, LanguageCode: languageCode, }
    );

    // The source language is required for sentiment analysis and
    // translation in the next step.
    const { Languages } = await comprehendClient.send(
        detectDominantLanguageCommand, 
    );

    const languageCode = Languages[0].LanguageCode;

    const detectSentimentCommand = new DetectSentimentCommand(
        { Text: source_text, LanguageCode: languageCode, }
    );
```

const { Sentiment } = await comprehendClient.send(detectSentimentCommand);

return {
    sentiment: Sentiment,
    language_code: languageCode,
};

import {
    DetectDocumentTextCommand,
    TextractClient,
} from "@aws-sdk/client-textract";

/**
 * Fetch the S3 object from the event and analyze it using Amazon Textract.
 *
 * @param {import("@types/aws-lambda").EventBridgeEvent<"Object Created">} eventBridgeS3Event
 */
export const handler = async (eventBridgeS3Event) => {
    const textractClient = new TextractClient();

    const detectDocumentTextCommand = new DetectDocumentTextCommand({
        Document: {
            S3Object: {
                Bucket: eventBridgeS3Event.bucket,
                Name: eventBridgeS3Event.object,
            },
        },
    });

    // Textract returns a list of blocks. A block can be a line, a page, word, etc.
    // Each block also contains geometry of the detected text.
    // For more information on the Block type, see https://docs.aws.amazon.com/textract/latest/dg/API_Block.html.
    const { Blocks } = await textractClient.send(detectDocumentTextCommand);

    // For the purpose of this example, we are only interested in words.
    const extractedWords = Blocks.filter((b) => b.BlockType === "WORD").map((b) => b.Text,
    )

    // Create an application to analyze customer feedback
    // 2006
import { PollyClient, SynthesizeSpeechCommand } from '@aws-sdk/client-polly';
import { S3Client } from '@aws-sdk/client-s3';
import { Upload } from '@aws-sdk/lib-storage';

/**
 * Synthesize an audio file from text.
 *
 * @param {{ bucket: string, translated_text: string, object: string}} sourceDestinationConfig
 */
export const handler = async (sourceDestinationConfig) => {
    const pollyClient = new PollyClient({});

    const synthesizeSpeechCommand = new SynthesizeSpeechCommand({
        Engine: "neural",
        Text: sourceDestinationConfig.translated_text,
        VoiceId: "Ruth",
        OutputFormat: "mp3",
    });

    const { AudioStream } = await pollyClient.send(synthesizeSpeechCommand);

    const audioKey = `${sourceDestinationConfig.object}.mp3`;

    // Store the audio file in S3.
    const s3Client = new S3Client();
    const upload = new Upload({
        client: s3Client,
        params: {
            Bucket: sourceDestinationConfig.bucket,
            Key: audioKey,
            Body: AudioStream,
            ContentType: "audio/mp3",
        },
    });

    await upload.done();
    return audioKey;
};
import {
  TranslateClient,
  TranslateTextCommand,
} from "@aws-sdk/client-translate";

/**
 * Translate the extracted text to English.
 *
 * @param {{ extracted_text: string, source_language_code: string}}
textAndSourceLanguage
 */
export const handler = async (textAndSourceLanguage) => {
  const translateClient = new TranslateClient({});

  const translateCommand = new TranslateTextCommand({
    SourceLanguageCode: textAndSourceLanguage.source_language_code,
    TargetLanguageCode: "en",
    Text: textAndSourceLanguage.extracted_text,
  });

  const { TranslatedText } = await translateClient.send(translateCommand);

  return { translated_text: TranslatedText };};

Services used in this example

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Ruby

SDK for Ruby

This example application analyzes and stores customer feedback cards. Specifically, it fulfills the need of a fictitious hotel in New York City. The hotel receives feedback from guests in various languages in the form of physical comment cards. That feedback is uploaded into
the app through a web client. After an image of a comment card is uploaded, the following steps occur:

- Text is extracted from the image using Amazon Textract.
- Amazon Comprehend determines the sentiment of the extracted text and its language.
- The extracted text is translated to English using Amazon Translate.
- Amazon Polly synthesizes an audio file from the extracted text.

The full app can be deployed with the AWS CDK. For source code and deployment instructions, see the project in [GitHub](https://github.com).

**Services used in this example**

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK](https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/integrating-lambda-with-sdk.html). This topic also includes information about getting started and details about previous SDK versions.

**Invoke a Lambda function from a browser**

The following code example shows how to invoke an AWS Lambda function from a browser.

**JavaScript**

**SDK for JavaScript (v2)**

You can create a browser-based application that uses an AWS Lambda function to update an Amazon DynamoDB table with user selections.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](https://github.com).

**Services used in this example**

- DynamoDB
You can create a browser-based application that uses an AWS Lambda function to update an Amazon DynamoDB table with user selections. This app uses AWS SDK for JavaScript v3.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use API Gateway to invoke a Lambda function

The following code examples show how to create an AWS Lambda function invoked by Amazon API Gateway.

Java

SDK for Java 2.x

Shows how to create an AWS Lambda function by using the Lambda Java runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create a Lambda function invoked by Amazon API Gateway that scans an Amazon DynamoDB table for work anniversaries and uses Amazon Simple Notification Service (Amazon SNS) to send a text message to your employees that congratulates them at their one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
• DynamoDB
• Lambda
• Amazon SNS

JavaScript

**SDK for JavaScript (v3)**

Shows how to create an AWS Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create a Lambda function invoked by Amazon API Gateway that scans an Amazon DynamoDB table for work anniversaries and uses Amazon Simple Notification Service (Amazon SNS) to send a text message to your employees that congratulates them at their one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](https://github.com). This example is also available in the [AWS SDK for JavaScript v3 developer guide](https://aws.amazon.com/sdk-for-javascript/).
• Create a REST resource that targets the Lambda function.
• Grant permission to let API Gateway invoke the Lambda function.
• Use the Requests package to send requests to the REST API.
• Clean up all resources created during the demo.

This example is best viewed on GitHub. For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

• API Gateway
• Lambda

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use Step Functions to invoke Lambda functions

The following code examples show how to create an AWS Step Functions state machine that invokes AWS Lambda functions in sequence.

Java

SDK for Java 2.x

Shows how to create an AWS serverless workflow by using AWS Step Functions and the AWS SDK for Java 2.x. Each workflow step is implemented using an AWS Lambda function.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

• DynamoDB
• Lambda
• Amazon SES
• Step Functions
JavaScript

**SDK for JavaScript (v3)**

Shows how to create an AWS serverless workflow by using AWS Step Functions and the AWS SDK for JavaScript. Each workflow step is implemented using an AWS Lambda function.

Lambda is a compute service that enables you to run code without provisioning or managing servers. Step Functions is a serverless orchestration service that lets you combine Lambda functions and other AWS services to build business-critical applications.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](https://github.com).

This example is also available in the [AWS SDK for JavaScript v3 developer guide](https://aws.amazon.com/).

**Services used in this example**

- DynamoDB
- Lambda
- Amazon SES
- Step Functions

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK](https://aws.amazon.com/). This topic also includes information about getting started and details about previous SDK versions.

**Use scheduled events to invoke a Lambda function**

The following code examples show how to create an AWS Lambda function invoked by an Amazon EventBridge scheduled event.

Java

**SDK for Java 2.x**

Shows how to create an Amazon EventBridge scheduled event that invokes an AWS Lambda function. Configure EventBridge to use a cron expression to schedule when the Lambda function is invoked. In this example, you create a Lambda function by using the Lambda Java
This example invokes different AWS services to perform a specific use case. This example demonstrates how to create an app that sends a mobile text message to your employees that congratulates them at the one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

**Services used in this example**

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

JavaScript

**SDK for JavaScript (v3)**

Shows how to create an Amazon EventBridge scheduled event that invokes an AWS Lambda function. Configure EventBridge to use a cron expression to schedule when the Lambda function is invoked. In this example, you create a Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create an app that sends a mobile text message to your employees that congratulates them at the one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

This example is also available in the AWS SDK for JavaScript v3 developer guide.

**Services used in this example**

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS
This example shows how to register an AWS Lambda function as the target of a scheduled Amazon EventBridge event. The Lambda handler writes a friendly message and the full event data to Amazon CloudWatch Logs for later retrieval.

- Deploys a Lambda function.
- Creates an EventBridge scheduled event and makes the Lambda function the target.
- Grants permission to let EventBridge invoke the Lambda function.
- Prints the latest data from CloudWatch Logs to show the result of the scheduled invocations.
- Cleans up all resources created during the demo.

This example is best viewed on GitHub. For complete source code and instructions on how to set up and run, see the full example on GitHub.

**Services used in this example**

- CloudWatch Logs
- EventBridge
- Lambda

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.
Lambda quotas

⚠️ Important

New AWS accounts have reduced concurrency and memory quotas. AWS raises these quotas automatically based on your usage.

Compute and storage

Lambda sets quotas for the amount of compute and storage resources that you can use to run and store functions. Quotas for concurrent executions and storage apply per AWS Region. Elastic network interface (ENI) quotas apply per virtual private cloud (VPC), regardless of Region. The following quotas can be increased from their default values. For more information, see Requesting a quota increase in the Service Quotas User Guide.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Default quota</th>
<th>Can be increased up to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrent executions</td>
<td>1,000</td>
<td>Tens of thousands</td>
</tr>
<tr>
<td>Storage for uploaded functions (.zip file archives) and layers. Each function version and layer version consumes storage.</td>
<td>75 GB</td>
<td>Terabytes</td>
</tr>
<tr>
<td>For best practices on managing your code storage, see Monitoring Lambda code storage in Serverless Land.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storage for functions defined as container images. These images are stored in Amazon ECR.</td>
<td>See Amazon ECR service quotas.</td>
<td></td>
</tr>
<tr>
<td>Elastic network interfaces per virtual private cloud (VPC)</td>
<td>250</td>
<td>Thousands</td>
</tr>
</tbody>
</table>
### Note
This quota is shared with other services, such as Amazon Elastic File System (Amazon EFS). See [Amazon VPC quotas](#).

For details on concurrency and how Lambda scales your function concurrency in response to traffic, see [Lambda function scaling](#).

## Function configuration, deployment, and execution

The following quotas apply to function configuration, deployment, and execution. Except as noted, they can't be changed.

### Note
The Lambda documentation, log messages, and console use the abbreviation MB (rather than MiB) to refer to 1,024 KB.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function memory allocation</td>
<td>128 MB to 10,240 MB, in 1-MB increments.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Lambda allocates CPU power in proportion to the amount of memory configured. You can increase or decrease the memory and CPU power allocated to your function using the Memory (MB)</td>
</tr>
<tr>
<td>Resource</td>
<td>Quota</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Function quota setting</td>
<td>At 1,769 MB, a function has the equivalent of one vCPU.</td>
</tr>
<tr>
<td>Function timeout</td>
<td>900 seconds (15 minutes)</td>
</tr>
<tr>
<td>Function environment variables</td>
<td>4 KB, for all environment variables associated with the function, in aggregate</td>
</tr>
<tr>
<td>Function resource-based policy</td>
<td>20 KB</td>
</tr>
<tr>
<td>Function layers</td>
<td>five layers</td>
</tr>
<tr>
<td>Function concurrency scaling limit</td>
<td>For each function, 1,000 execution environments every 10 seconds</td>
</tr>
<tr>
<td>Invocation payload (request and response)</td>
<td>6 MB each for request and response (synchronous)</td>
</tr>
<tr>
<td></td>
<td>20 MB for each streamed response (Synchronous. The payload size for streamed responses can be increased from default values. Contact AWS Support to inquire further.)</td>
</tr>
<tr>
<td></td>
<td>256 KB (asynchronous)</td>
</tr>
<tr>
<td>Bandwidth for streamed responses</td>
<td>Uncapped for the first 6 MB of your function's response</td>
</tr>
<tr>
<td></td>
<td>For responses larger than 6 MB, 2MBps for the remainder of the response</td>
</tr>
</tbody>
</table>
### Resource Quota

<table>
<thead>
<tr>
<th>Resource</th>
<th>Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Deployment package (.zip file archive) size</strong></td>
<td>50 MB (zipped, for direct upload)</td>
</tr>
<tr>
<td></td>
<td>250 MB (unzipped)</td>
</tr>
<tr>
<td></td>
<td>This quota applies to all the files you upload, including layers and custom runtimes.</td>
</tr>
<tr>
<td></td>
<td>3 MB (console editor)</td>
</tr>
<tr>
<td><strong>Container image settings size</strong></td>
<td>16 KB</td>
</tr>
<tr>
<td><strong>Container image</strong> code package size</td>
<td>10 GB (maximum uncompressed image size, including all layers)</td>
</tr>
<tr>
<td><strong>Test events (console editor)</strong></td>
<td>10</td>
</tr>
<tr>
<td><strong>/tmp directory storage</strong></td>
<td>Between 512 MB and 10,240 MB, in 1-MB increments</td>
</tr>
<tr>
<td><strong>File descriptors</strong></td>
<td>1,024</td>
</tr>
<tr>
<td><strong>Execution processes/threads</strong></td>
<td>1,024</td>
</tr>
</tbody>
</table>

### Lambda API requests

The following quotas are associated with Lambda API requests.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invocation requests per function per Region (synchronous)</td>
<td>Each instance of your execution environment can serve up to 10 requests per second. In other words, the total invocation limit is 10 times your concurrency limit. See <a href="https://docs.aws.amazon.com/lambda/latest/dg/func-concurrency.html">Lambda function scaling</a>.</td>
</tr>
<tr>
<td>Resource</td>
<td>Quota</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Invocation requests per function per Region (asynchronous)</td>
<td>Each instance of your execution environment can serve an unlimited number of requests. In other words, the total invocation limit is based only on concurrency available to your function. See Lambda function scaling.</td>
</tr>
<tr>
<td>Invocation requests per function version or alias (requests per second)</td>
<td>10 x allocated provisioned concurrency</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Note</strong></td>
<td>This quota applies only to functions that use provisioned concurrency.</td>
</tr>
<tr>
<td>GetFunction API requests</td>
<td>100 requests per second</td>
</tr>
<tr>
<td>GetPolicy API requests</td>
<td>15 requests per second</td>
</tr>
<tr>
<td>Remainder of the control plane API requests (excludes invocation, GetFunction, and GetPolicy requests)</td>
<td>15 requests per second</td>
</tr>
</tbody>
</table>

**Other services**

Quotas for other services, such as AWS Identity and Access Management (IAM), Amazon CloudFront (Lambda@Edge), and Amazon Virtual Private Cloud (Amazon VPC), can impact your Lambda functions. For more information, see AWS service quotas in the Amazon Web Services General Reference, and Using AWS Lambda with other services.
AWS Glossary

For the latest AWS terminology, see the AWS glossary in the AWS Glossary Reference.
API reference

This section contains the AWS Lambda API Reference documentation. When making the API calls, you will need to authenticate your request by providing a signature. AWS Lambda supports signature version 4. For more information, see Signature Version 4 signing process in the Amazon Web Services General Reference.

For an overview of the service, see What is AWS Lambda?

You can use the AWS CLI to explore the AWS Lambda API. This guide provides several tutorials that use the AWS CLI.

Topics

- Actions
- Data Types

Actions

The following actions are supported:

- AddLayerVersionPermission
- AddPermission
- CreateAlias
- CreateCodeSigningConfig
- CreateEventSourceMapping
- CreateFunction
- CreateFunctionUrlConfig
- DeleteAlias
- DeleteCodeSigningConfig
- DeleteEventSourceMapping
- DeleteFunction
- DeleteFunctionCodeSigningConfig
- DeleteFunctionConcurrency
• DeleteFunctionEventInvokeConfig
• DeleteFunctionUrlConfig
• DeleteLayerVersion
• DeleteProvisionedConcurrencyConfig
• GetAccountSettings
• GetAlias
• GetCodeSigningConfig
• GetEventSourceMapping
• GetFunction
• GetFunctionCodeSigningConfig
• GetFunctionConcurrency
• GetFunctionConfiguration
• GetFunctionEventInvokeConfig
• GetFunctionUrlConfig
• GetLayerVersion
• GetLayerVersionByArn
• GetLayerVersionPolicy
• GetPolicy
• GetProvisionedConcurrencyConfig
• GetRuntimeManagementConfig
• Invoke
• InvokeAsync
• InvokeWithResponseStream
• ListAliases
• ListCodeSigningConfigs
• ListEventSourceMappings
• ListFunctionEventInvokeConfigs
• ListFunctions
• ListFunctionsByCodeSigningConfig
• ListFunctionUrlConfigs
- ListLayers
- ListLayerVersions
- List Provisioned Concurrency Configs
- ListTags
- ListVersionsByFunction
- PublishLayerVersion
- PublishVersion
- PutFunctionCodeSigningConfig
- PutFunctionConcurrency
- PutFunctionEventInvokeConfig
- Put Provisioned Concurrency Config
- PutRuntimeManagementConfig
- RemoveLayerVersionPermission
- RemovePermission
- TagResource
- UntagResource
- UpdateAlias
- UpdateCodeSigningConfig
- UpdateEventSourceMapping
- UpdateFunctionCode
- UpdateFunctionConfiguration
- UpdateFunctionEventInvokeConfig
- UpdateFunctionUrlConfig
AddLayerVersionPermission

Adds permissions to the resource-based policy of a version of an AWS Lambda layer. Use this action to grant layer usage permission to other accounts. You can grant permission to a single account, all accounts in an organization, or all AWS accounts.

To revoke permission, call RemoveLayerVersionPermission with the statement ID that you specified when you added it.

Request Syntax

```
POST /2018-10-31/layers/{LayerName}/versions/{VersionNumber}/policy?RevisionId={RevisionId}
HTTP/1.1
Content-type: application/json

{
    "Action": "string",
    "OrganizationId": "string",
    "Principal": "string",
    "StatementId": "string"
}
```

URI Request Parameters

The request uses the following URI parameters.

LayerName

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+::\d{12}:layer:[a-zA-Z0-9-\_]+)+|[a-zA-Z0-9-\_]+

Required: Yes

RevisionId

Only update the policy if the revision ID matches the ID specified. Use this option to avoid modifying a policy that has changed since you last read it.
**VersionNumber**

The version number.

Required: Yes

**Request Body**

The request accepts the following data in JSON format.

**Action**

The API action that grants access to the layer. For example, `lambda:GetLayerVersion`.

Type: String

Length Constraints: Maximum length of 22.

Pattern: `lambda:GetLayerVersion`

Required: Yes

**OrganizationId**

With the principal set to *, grant permission to all accounts in the specified organization.

Type: String

Length Constraints: Maximum length of 34.

Pattern: `o-[a-zA-Z0-9]{10,32}`

Required: No

**Principal**

An account ID, or * to grant layer usage permission to all accounts in an organization, or all AWS accounts (if `organizationId` is not specified). For the last case, make sure that you really do want all AWS accounts to have usage permission to this layer.

Type: String

Pattern: `\d{12}|\*|arn:(aws[a-zA-Z-]*):iam::\d{12}:root`
StatementId

An identifier that distinguishes the policy from others on the same layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 100.

Pattern: ([a-zA-Z0-9-_]*)

Required: Yes

Response Syntax

HTTP/1.1 201
Content-type: application/json

{
    "RevisionId": "string",
    "Statement": "string"
}

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

RevisionId

A unique identifier for the current revision of the policy.

Type: String

Statement

The permission statement.

Type: String
Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

PolicyLengthExceededException

The permissions policy for the resource is too large. For more information, see Lambda quotas.

HTTP Status Code: 400

PreconditionFailedException

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429
See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
AddPermission

Grants an AWS service, AWS account, or AWS organization permission to use a function. You can apply the policy at the function level, or specify a qualifier to restrict access to a single version or alias. If you use a qualifier, the invoker must use the full Amazon Resource Name (ARN) of that version or alias to invoke the function. Note: Lambda does not support adding policies to version $LATEST.

To grant permission to another account, specify the account ID as the Principal. To grant permission to an organization defined in AWS Organizations, specify the organization ID as the PrincipalOrgID. For AWS services, the principal is a domain-style identifier that the service defines, such as s3.amazonaws.com or sns.amazonaws.com. For AWS services, you can also specify the ARN of the associated resource as the SourceArn. If you grant permission to a service principal without specifying the source, other accounts could potentially configure resources in their account to invoke your Lambda function.

This operation adds a statement to a resource-based permissions policy for the function. For more information about function policies, see [Using resource-based policies for Lambda](#).

Request Syntax

POST /2015-03-31/functions/FunctionName/policy?Qualifier=Qualifier HTTP/1.1
Content-type: application/json

```
{
   "Action": "string",
   "EventSourceToken": "string",
   "FunctionUrlAuthType": "string",
   "Principal": "string",
   "PrincipalOrgID": "string",
   "RevisionId": "string",
   "SourceAccount": "string",
   "SourceArn": "string",
   "StatementId": "string"
}
```

URI Request Parameters

The request uses the following URI parameters.
**FunctionName**

The name of the Lambda function, version, or alias.

**Name formats**

- **Function name** – my-function (name-only), my-function:v1 (with alias).
- **Partial ARN** – 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

**Length Constraints**: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*?):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-._]+)(:\$LATEST|[a-zA-Z0-9-._]+))?

Required: Yes

**Qualifier**

Specify a version or alias to add permissions to a published version of the function.

**Length Constraints**: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9-._-]+)

**Request Body**

The request accepts the following data in JSON format.

**Action**

The action that the principal can use on the function. For example, lambda:InvokeFunction or lambda:GetFunction.

Type: String

Pattern: (lambda:[*]|lambda:[a-zA-Z]+|[*])
**Required:** Yes

### EventSourceToken

For Alexa Smart Home functions, a token that the invoker must supply.

**Type:** String

**Length Constraints:** Minimum length of 0. Maximum length of 256.

**Pattern:** [a-zA-Z0-9._-]+

**Required:** No

### FunctionUrlAuthType

The type of authentication that your function URL uses. Set to AWS_IAM if you want to restrict access to authenticated users only. Set to NONE if you want to bypass IAM authentication to create a public endpoint. For more information, see [Security and auth model for Lambda function URLs](#).

**Type:** String

**Valid Values:** NONE | AWS_IAM

**Required:** No

### Principal

The AWS service or AWS account that invokes the function. If you specify a service, use SourceArn or SourceAccount to limit who can invoke the function through that service.

**Type:** String

**Pattern:** [^\s]+

**Required:** Yes

### PrincipalOrgID

The identifier for your organization in AWS Organizations. Use this to grant permissions to all the AWS accounts under this organization.

**Type:** String
**RevisionId**

Update the policy only if the revision ID matches the ID that's specified. Use this option to avoid modifying a policy that has changed since you last read it.

Type: String

Required: No

**SourceAccount**

For AWS service, the ID of the AWS account that owns the resource. Use this together with SourceArn to ensure that the specified account owns the resource. It is possible for an Amazon S3 bucket to be deleted by its owner and recreated by another account.

Type: String

Length Constraints: Maximum length of 12.

Pattern: \d{12}

Required: No

**SourceArn**

For AWS services, the ARN of the AWS resource that invokes the function. For example, an Amazon S3 bucket or Amazon SNS topic.

Note that Lambda configures the comparison using the StringLike operator.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*)\([a-zA-Z0-9\-]+([a-z]{2}(-gov)?-[a-z]+-[\d\w])?:([a-z]{2}(-gov)?-[a-z]+-[\d\w])?:(.*)

Required: No

**StatementId**

A statement identifier that differentiates the statement from others in the same policy.
Type: String

Length Constraints: Minimum length of 1. Maximum length of 100.

Pattern: ([a-zA-Z0-9-_]*)

Required: Yes

Response Syntax

HTTP/1.1 201
Content-type: application/json

{
   "Statement": "string"
}

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

Statement

The permission statement that's added to the function policy.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

PolicyLengthExceededException

The permissions policy for the resource is too large. For more information, see Lambda quotas.

HTTP Status Code: 400
PreconditionFailedException

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
CreateAlias

Creates an alias for a Lambda function version. Use aliases to provide clients with a function identifier that you can update to invoke a different version.

You can also map an alias to split invocation requests between two versions. Use the RoutingConfig parameter to specify a second version and the percentage of invocation requests that it receives.

Request Syntax

```json
POST /2015-03-31/functions/{FunctionName}/aliases HTTP/1.1
Content-type: application/json

{
  "Description": "string",
  "FunctionVersion": "string",
  "Name": "string",
  "RoutingConfig": {
    "AdditionalVersionWeights": {
      "string": number
    }
  }
}
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.
Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z-A-Z0-9-\_]+)(:(\$LATEST|[a-zA-Z0-9-\_]+))? 
Required: Yes

Request Body

The request accepts the following data in JSON format.

**Description**

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

**FunctionVersion**

The function version that the alias invokes.

Type: String


Pattern: (\$LATEST|[^0-9]+)

Required: Yes

**Name**

The name of the alias.

Type: String


Pattern: (?![^0-9]+$)([a-zA-Z0-9-\_]*)

Required: Yes

**RoutingConfig**

The *routing configuration* of the alias.

Type: *AliasRoutingConfiguration* object

Required: No

**Response Syntax**

HTTP/1.1 201
Content-type: application/json

```
{
    "AliasArn": "string",
    "Description": "string",
    "FunctionVersion": "string",
    "Name": "string",
    "RevisionId": "string",
    "RoutingConfig": {
        "AdditionalVersionWeights": {
            "string": number
        }
    }
}
```

**Response Elements**

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

**AliasArn**

The Amazon Resource Name (ARN) of the alias.

Type: String

Pattern: \*arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_=]+:(\$LATEST|\[a-zA-Z0-9-_=]+)?
**Description**

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

**FunctionVersion**

The function version that the alias invokes.

Type: String


Pattern: (\$LATEST|[0-9]+)

**Name**

The name of the alias.

Type: String


Pattern: (?!^[0-9]+$)([a-zA-Z0-9-]+)

**RevisionId**

A unique identifier that changes when you update the alias.

Type: String

**RoutingConfig**

The routing configuration of the alias.

Type: AliasRoutingConfiguration object

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.
HTTP Status Code: 400

**ResourceConflictException**

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
CreateCodeSigningConfig

Creates a code signing configuration. A code signing configuration defines a list of allowed signing profiles and defines the code-signing validation policy (action to be taken if deployment validation checks fail).

Request Syntax

POST /2020-04-22/code-signing-configs/ HTTP/1.1
Content-type: application/json

{
    "AllowedPublishers": {
        "SigningProfileVersionArns": [ "string" ]
    },
    "CodeSigningPolicies": {
        "UntrustedArtifactOnDeployment": "string"
    },
    "Description": "string"
}

URI Request Parameters

The request does not use any URI parameters.

Request Body

The request accepts the following data in JSON format.

**AllowedPublishers**

Signing profiles for this code signing configuration.

Type: AllowedPublishers object

Required: Yes

**CodeSigningPolicies**

The code signing policies define the actions to take if the validation checks fail.

Type: CodeSigningPolicies object
Required: No

**Description**

Descriptive name for this code signing configuration.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

**Response Syntax**

HTTP/1.1 201
Content-type: application/json

```json
{
    "CodeSigningConfig": {
        "AllowedPublishers": {
            "SigningProfileVersionArns": [ "string" ]
        },
        "CodeSigningConfigArn": "string",
        "CodeSigningConfigId": "string",
        "CodeSigningPolicies": {
            "UntrustedArtifactOnDeployment": "string"
        },
        "Description": "string",
        "LastModified": "string"
    }
}
```

**Response Elements**

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

**CodeSigningConfig**

The code signing configuration.

Type: [CodeSigningConfig](#) object
Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
CreateEventSourceMapping

Creates a mapping between an event source and an AWS Lambda function. Lambda reads items from the event source and invokes the function.

For details about how to configure different event sources, see the following topics.

- Amazon DynamoDB Streams
- Amazon Kinesis
- Amazon SQS
- Amazon MQ and RabbitMQ
- Amazon MSK
- Apache Kafka
- Amazon DocumentDB

The following error handling options are available only for stream sources (DynamoDB and Kinesis):

- BisectBatchOnFunctionError – If the function returns an error, split the batch in two and retry.
- DestinationConfig – Send discarded records to an Amazon SQS queue or Amazon SNS topic.
- MaximumRecordAgeInSeconds – Discard records older than the specified age. The default value is infinite (-1). When set to infinite (-1), failed records are retried until the record expires.
- MaximumRetryAttempts – Discard records after the specified number of retries. The default value is infinite (-1). When set to infinite (-1), failed records are retried until the record expires.
- ParallelizationFactor – Process multiple batches from each shard concurrently.

For information about which configuration parameters apply to each event source, see the following topics.

- Amazon DynamoDB Streams
- Amazon Kinesis
- Amazon SQS
- Amazon MQ and RabbitMQ
• Amazon MSK
• Apache Kafka
• Amazon DocumentDB

Request Syntax

POST /2015-03-31/event-source-mappings/ HTTP/1.1
Content-type: application/json

{
   "AmazonManagedKafkaEventSourceConfig": {
      "ConsumerGroupId": "string"
   },
   "BatchSize": number,
   "BisectBatchOnFunctionError": boolean,
   "DestinationConfig": {
      "OnFailure": {
         "Destination": "string"
      },
      "OnSuccess": {
         "Destination": "string"
      }
   },
   "DocumentDBEventSourceConfig": {
      "CollectionName": "string",
      "DatabaseName": "string",
      "FullDocument": "string"
   },
   "Enabled": boolean,
   "EventSourceArn": "string",
   "FilterCriteria": {
      "Filters": [
         {
            "Pattern": "string"
         }
      ]
   },
   "FunctionName": "string",
   "FunctionResponseTypes": [ "string" ],
   "MaximumBatchingWindowInSeconds": number,
   "MaximumRecordAgeInSeconds": number,
   "MaximumRetryAttempts": number,
}
"ParallelizationFactor": number,
"Queues": [ "string" ],
"ScalingConfig": {
    "MaximumConcurrency": number
},
"SelfManagedEventSource": {
    "Endpoints": {
        "string": [ "string"
    }
},
"SelfManagedKafkaEventSourceConfig": {
    "ConsumerGroupId": "string"
},
"SourceAccessConfigurations": [
    {
        "Type": "string",
        "URI": "string"
    }
],
"StartingPosition": "string",
"StartingPositionTimestamp": number,
"Topics": [ "string" ],
"TumblingWindowInSeconds": number

**URI Request Parameters**

The request does not use any URI parameters.

**Request Body**

The request accepts the following data in JSON format.

**AmazonManagedKafkaEventSourceConfig**

Specific configuration settings for an Amazon Managed Streaming for Apache Kafka (Amazon MSK) event source.

Type: AmazonManagedKafkaEventSourceConfig object

Required: No
**BatchSize**

The maximum number of records in each batch that Lambda pulls from your stream or queue and sends to your function. Lambda passes all of the records in the batch to the function in a single call, up to the payload limit for synchronous invocation (6 MB).

- **Amazon Kinesis** – Default 100. Max 10,000.
- **Amazon DynamoDB Streams** – Default 100. Max 10,000.
- **Amazon Simple Queue Service** – Default 10. For standard queues the max is 10,000. For FIFO queues the max is 10.
- **Amazon Managed Streaming for Apache Kafka** – Default 100. Max 10,000.
- **Self-managed Apache Kafka** – Default 100. Max 10,000.
- **Amazon MQ (ActiveMQ and RabbitMQ)** – Default 100. Max 10,000.
- **DocumentDB** – Default 100. Max 10,000.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10,000.

Required: No

**BisectBatchOnFunctionError**

(Kinesis and DynamoDB Streams only) If the function returns an error, split the batch in two and retry.

Type: Boolean

Required: No

**DestinationConfig**

(Kinesis, DynamoDB Streams, Amazon MSK, and self-managed Kafka only) A configuration object that specifies the destination of an event after Lambda processes it.

Type: `DestinationConfig` object

Required: No

**DocumentDBEventSourceConfig**

Specific configuration settings for a DocumentDB event source.

Type: `DocumentDBEventSourceConfig` object
**Enabled**

When true, the event source mapping is active. When false, Lambda pauses polling and invocation.

Default: True

Type: Boolean

Required: No

**EventSourceArn**

The Amazon Resource Name (ARN) of the event source.

- **Amazon Kinesis** – The ARN of the data stream or a stream consumer.
- **Amazon DynamoDB Streams** – The ARN of the stream.
- **Amazon Simple Queue Service** – The ARN of the queue.
- **Amazon Managed Streaming for Apache Kafka** – The ARN of the cluster or the ARN of the VPC connection (for cross-account event source mappings).
- **Amazon MQ** – The ARN of the broker.
- **Amazon DocumentDB** – The ARN of the DocumentDB change stream.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*)\([a-zA-Z0-9\-]\)+([a-z]{2}(-gov)?-[a-z]+-\d{1})?:(\d{12})?:(.*)`

Required: No

**FilterCriteria**

An object that defines the filter criteria that determine whether Lambda should process an event. For more information, see [Lambda event filtering](#).

Type: **FilterCriteria** object

Required: No

**FunctionName**

The name of the Lambda function.
Name formats

- **Function name** – MyFunction.
- **Partial ARN** – `123456789012:function:MyFunction`.

The length constraint applies only to the full ARN. If you specify only the function name, it’s limited to 64 characters in length.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-\_]*)?(:($LATEST|[a-zA-Z0-9-\_]*)?)`?

Required: Yes

**FunctionResponseTypes**

(Kinesis, DynamoDB Streams, and Amazon SQS) A list of current response type enums applied to the event source mapping.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 1 item.

Valid Values: ReportBatchItemFailures

Required: No

**MaximumBatchingWindowInSeconds**

The maximum amount of time, in seconds, that Lambda spends gathering records before invoking the function. You can configure `MaximumBatchingWindowInSeconds` to any value from 0 seconds to 300 seconds in increments of seconds.

For streams and Amazon SQS event sources, the default batching window is 0 seconds. For Amazon MSK, Self-managed Apache Kafka, Amazon MQ, and DocumentDB event sources, the default batching window is 500 ms. Note that because you can only change `MaximumBatchingWindowInSeconds` in increments of seconds, you cannot revert back to
the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

Related setting: For streams and Amazon SQS event sources, when you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

Required: No

**MaximumRecordAgeInSeconds**

(Kinesis and DynamoDB Streams only) Discard records older than the specified age. The default value is infinite (-1).

Type: Integer


Required: No

**MaximumRetryAttempts**

(Kinesis and DynamoDB Streams only) Discard records after the specified number of retries. The default value is infinite (-1). When set to infinite (-1), failed records are retried until the record expires.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

Required: No

**ParallelizationFactor**

(Kinesis and DynamoDB Streams only) The number of batches to process from each shard concurrently.

Type: Integer


Required: No
**Queues**

(MQ) The name of the Amazon MQ broker destination queue to consume.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 1000.

Pattern: \[\s\S\]*

Required: No

**ScalingConfig**

(Amazon SQS only) The scaling configuration for the event source. For more information, see [Configuring maximum concurrency for Amazon SQS event sources](#).

Type: `ScalingConfig` object

Required: No

**SelfManagedEventSource**

The self-managed Apache Kafka cluster to receive records from.

Type: `SelfManagedEventSource` object

Required: No

**SelfManagedKafkaEventSourceConfig**

Specific configuration settings for a self-managed Apache Kafka event source.

Type: `SelfManagedKafkaEventSourceConfig` object

Required: No

**SourceAccessConfigurations**

An array of authentication protocols or VPC components required to secure your event source.

Type: Array of `SourceAccessConfiguration` objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.
### StartingPosition

The position in a stream from which to start reading. Required for Amazon Kinesis and Amazon DynamoDB Stream event sources. AT_TIMESTAMP is supported only for Amazon Kinesis streams, Amazon DocumentDB, Amazon MSK, and self-managed Apache Kafka.

**Type:** String

**Valid Values:** TRIM_HORIZON | LATEST | AT_TIMESTAMP

**Required:** No

### StartingPositionTimestamp

With `StartingPosition` set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds. StartingPositionTimestamp cannot be in the future.

**Type:** Timestamp

**Required:** No

### Topics

The name of the Kafka topic.

**Type:** Array of strings

**Array Members:** Fixed number of 1 item.

**Length Constraints:** Minimum length of 1. Maximum length of 249.

**Pattern:** ^[^.][-,a-zA-Z0-9\-_.]+$

**Required:** No

### TumblingWindowInSeconds

(Kinesis and DynamoDB Streams only) The duration in seconds of a processing window for DynamoDB and Kinesis Streams event sources. A value of 0 seconds indicates no tumbling window.

**Type:** Integer

**Valid Range:** Minimum value of 0. Maximum value of 900.
Required: No

Response Syntax

HTTP/1.1 202
Content-type: application/json

{
    "AmazonManagedKafkaEventSourceConfig": {
        "ConsumerGroupId": "string"
    },
    "BatchSize": number,
    "BisectBatchOnFunctionError": boolean,
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "DocumentDBEventSourceConfig": {
        "CollectionName": "string",
        "DatabaseName": "string",
        "FullDocument": "string"
    },
    "EventSourceArn": "string",
    "FilterCriteria": {
        "Filters": [
            {
                "Pattern": "string"
            }
        ]
    },
    "FunctionArn": "string",
    "FunctionResponseTypes": [ "string" ],
    "LastModified": number,
    "LastProcessingResult": "string",
    "MaximumBatchingWindowInSeconds": number,
    "MaximumRecordAgeInSeconds": number,
    "MaximumRetryAttempts": number,
    "ParallelizationFactor": number,
    "Queues": [ "string" ]
}
"ScalingConfig": {
  "MaximumConcurrency": number
},
"SelfManagedEventSource": {
  "Endpoints": {
    "string": [ "string" ]
  }
},
"SelfManagedKafkaEventSourceConfig": {
  "ConsumerGroupId": "string"
},
"SourceAccessConfigurations": [
  {
    "Type": "string",
    "URI": "string"
  }
],
"StartingPosition": "string",
"StartingPositionTimestamp": number,
"State": "string",
"StateTransitionReason": "string",
"Topics": [ "string" ],
"TumblingWindowInSeconds": number,
"UUID": "string"
}

Response Elements

If the action is successful, the service sends back an HTTP 202 response.

The following data is returned in JSON format by the service.

**AmazonManagedKafkaEventSourceConfig**

Specific configuration settings for an Amazon Managed Streaming for Apache Kafka (Amazon MSK) event source.

Type: AmazonManagedKafkaEventSourceConfig object

**BatchSize**

The maximum number of records in each batch that Lambda pulls from your stream or queue and sends to your function. Lambda passes all of the records in the batch to the function in a single call, up to the payload limit for synchronous invocation (6 MB).
Default value: Varies by service. For Amazon SQS, the default is 10. For all other services, the default is 100.

Related setting: When you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

**BisectBatchOnFunctionError**

(Kinesis and DynamoDB Streams only) If the function returns an error, split the batch in two and retry. The default value is false.

Type: Boolean

**DestinationConfig**

(Kinesis and DynamoDB Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: DestinationConfig object

**DocumentDBEventSourceConfig**

Specific configuration settings for a DocumentDB event source.

Type: DocumentDBEventSourceConfig object

**EventSourceArn**

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):(\[a-zA-Z0-9\-\])+:([a-zA-Z]{2}(-gov)?-[a-zA-Z]+-\d{1})?:(\d{12})?:(.*)

**FilterCriteria**

An object that defines the filter criteria that determine whether Lambda should process an event. For more information, see Lambda event filtering.

Type: FilterCriteria object
**FunctionArn**

The ARN of the Lambda function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-[\d{1}]:\d{12}:function:[a-zA-Z0-9-\_]+(:($LATEST|a-zA-Z0-9-\_\_))?

**FunctionResponseTypes**

(Kinesis, DynamoDB Streams, and Amazon SQS) A list of current response type enums applied to the event source mapping.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 1 item.

Valid Values: ReportBatchItemFailures

**LastModified**

The date that the event source mapping was last updated or that its state changed, in Unix time seconds.

Type: Timestamp

**LastProcessingResult**

The result of the last Lambda invocation of your function.

Type: String

**MaximumBatchingWindowInSeconds**

The maximum amount of time, in seconds, that Lambda spends gathering records before invoking the function. You can configure MaximumBatchingWindowInSeconds to any value from 0 seconds to 300 seconds in increments of seconds.

For streams and Amazon SQS event sources, the default batching window is 0 seconds. For Amazon MSK, Self-managed Apache Kafka, Amazon MQ, and DocumentDB event sources, the default batching window is 500 ms. Note that because you can only change MaximumBatchingWindowInSeconds in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.
Related setting: For streams and Amazon SQS event sources, when you set `BatchSize` to a value greater than 10, you must set `MaximumBatchingWindowInSeconds` to at least 1.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

**MaximumRecordAgeInSeconds**

(Kinesis and DynamoDB Streams only) Discard records older than the specified age. The default value is -1, which sets the maximum age to infinite. When the value is set to infinite, Lambda never discards old records.

Type: Integer


**MaximumRetryAttempts**

(Kinesis and DynamoDB Streams only) Discard records after the specified number of retries. The default value is -1, which sets the maximum number of retries to infinite. When `MaximumRetryAttempts` is infinite, Lambda retries failed records until the record expires in the event source.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

**ParallelizationFactor**

(Kinesis and DynamoDB Streams only) The number of batches to process concurrently from each shard. The default value is 1.

Type: Integer

Queues

(Amazon MQ) The name of the Amazon MQ broker destination queue to consume.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 1000.

Pattern: [\s\S]*

ScalingConfig

(Amazon SQS only) The scaling configuration for the event source. For more information, see Configuring maximum concurrency for Amazon SQS event sources.

Type: ScalingConfig object

SelfManagedEventSource

The self-managed Apache Kafka cluster for your event source.

Type: SelfManagedEventSource object

SelfManagedKafkaEventSourceConfig

Specific configuration settings for a self-managed Apache Kafka event source.

Type: SelfManagedKafkaEventSourceConfig object

SourceAccessConfigurations

An array of the authentication protocol, VPC components, or virtual host to secure and define your event source.

Type: Array of SourceAccessConfiguration objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

StartingPosition

The position in a stream from which to start reading. Required for Amazon Kinesis and Amazon DynamoDB Stream event sources. AT_TIMESTAMP is supported only for Amazon Kinesis streams, Amazon DocumentDB, Amazon MSK, and self-managed Apache Kafka.

Type: String
Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

**StartingPositionTimestamp**

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds. StartingPositionTimestamp cannot be in the future.

Type: Timestamp

**State**

The state of the event source mapping. It can be one of the following: Creating, Enabling, Enabled, Disabling, Disabled, Updating, or Deleting.

Type: String

**StateTransitionReason**

Indicates whether a user or Lambda made the last change to the event source mapping.

Type: String

**Topics**

The name of the Kafka topic.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 249.

Pattern: `^[^.]([^a-zA-Z0-9\-_.]+)$`

**TumblingWindowInSeconds**

(Kinesis and DynamoDB Streams only) The duration in seconds of a processing window for DynamoDB and Kinesis Streams event sources. A value of 0 seconds indicates no tumbling window.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

**UUID**

The identifier of the event source mapping.
Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
CreateFunction

Creates a Lambda function. To create a function, you need a deployment package and an execution role. The deployment package is a .zip file archive or container image that contains your function code. The execution role grants the function permission to use AWS services, such as Amazon CloudWatch Logs for log streaming and AWS X-Ray for request tracing.

If the deployment package is a container image, then you set the package type to Image. For a container image, the code property must include the URI of a container image in the Amazon ECR registry. You do not need to specify the handler and runtime properties.

If the deployment package is a .zip file archive, then you set the package type to Zip. For a .zip file archive, the code property specifies the location of the .zip file. You must also specify the handler and runtime properties. The code in the deployment package must be compatible with the target instruction set architecture of the function (x86-64 or aarch64). If you do not specify the architecture, then the default value is x86-64.

When you create a function, Lambda provisions an instance of the function and its supporting resources. If your function connects to a VPC, this process can take a minute or so. During this time, you can't invoke or modify the function. The State, StateReason, and StateReasonCode fields in the response from GetFunctionConfiguration indicate when the function is ready to invoke. For more information, see Lambda function states.

A function has an unpublished version, and can have published versions and aliases. The unpublished version changes when you update your function's code and configuration. A published version is a snapshot of your function code and configuration that can't be changed. An alias is a named resource that maps to a version, and can be changed to map to a different version. Use the Publish parameter to create version 1 of your function from its initial configuration.

The other parameters let you configure version-specific and function-level settings. You can modify version-specific settings later with UpdateFunctionConfiguration. Function-level settings apply to both the unpublished and published versions of the function, and include tags (TagResource) and per-function concurrency limits (PutFunctionConcurrency).

You can use code signing if your deployment package is a .zip file archive. To enable code signing for this function, specify the ARN of a code-signing configuration. When a user attempts to deploy a code package with UpdateFunctionCode, Lambda checks that the code package has a valid signature from a trusted publisher. The code-signing configuration includes set of signing profiles, which define the trusted publishers for this function.
If another AWS account or an AWS service invokes your function, use `AddPermission` to grant permission by creating a resource-based AWS Identity and Access Management (IAM) policy. You can grant permissions at the function level, on a version, or on an alias.

To invoke your function directly, use `Invoke`. To invoke your function in response to events in other AWS services, create an event source mapping (`CreateEventSourceMapping`), or configure a function trigger in the other service. For more information, see [Invoking Lambda functions](#).

### Request Syntax

```
POST /2015-03-31/functions HTTP/1.1
Content-type: application/json

{
  "Architectures": [ "string" ],
  "Code": {
    "ImageUri": "string",
    "S3Bucket": "string",
    "S3Key": "string",
    "S3ObjectVersion": "string",
    "ZipFile": "blob"
  },
  "CodeSigningConfigArn": "string",
  "DeadLetterConfig": {
    "TargetArn": "string"
  },
  "Description": "string",
  "Environment": {
    "Variables": {
      "string": "string"
    }
  },
  "EphemeralStorage": {
    "Size": number
  },
  "FileSystemConfigs": [
    {
      "Arn": "string",
      "LocalMountPath": "string"
    }
  ],
  "FunctionName": "string",
  "Handler": "string"
}
```
"ImageConfig": {
    "Command": [ "string" ],
    "EntryPoint": [ "string" ],
    "WorkingDirectory": "string"
},
"KMSKeyArn": "string",
"Layers": [ "string" ],
"LoggingConfig": {
    "ApplicationLogLevel": "string",
    "LogFormat": "string",
    "LogGroup": "string",
    "SystemLogLevel": "string"
},
"MemorySize": number,
"PackageType": "string",
"Publish": boolean,
"Role": "string",
"Runtime": "string",
"SnapStart": {
    "ApplyOn": "string"
},
"Tags": {
    "string": "string"
},
"Timeout": number,
"TracingConfig": {
    "Mode": "string"
},
"VpcConfig": {
    "Ipv6AllowedForDualStack": boolean,
    "SecurityGroupIds": [ "string" ],
    "SubnetIds": [ "string" ]
}

### URI Request Parameters

The request does not use any URI parameters.

### Request Body

The request accepts the following data in JSON format.
Architectures

The instruction set architecture that the function supports. Enter a string array with one of the valid values (arm64 or x86_64). The default value is x86_64.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: x86_64 | arm64

Required: No

Code

The code for the function.

Type: FunctionCode object

Required: Yes

CodeSigningConfigArn

To enable code signing for this function, specify the ARN of a code-signing configuration. A code-signing configuration includes a set of signing profiles, which define the trusted publishers for this function.

Type: String

Length Constraints: Maximum length of 200.

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}((-gov)|(-iso(b?)))[-][a-z][-\d{1}:\d{12}:code-signing-config:csc-[-a-z0-9]{17}]

Required: No

DeadLetterConfig

A dead-letter queue configuration that specifies the queue or topic where Lambda sends asynchronous events when they fail processing. For more information, see Dead-letter queues.

Type: DeadLetterConfig object

Required: No
**Description**

A description of the function.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

**Environment**

Environment variables that are accessible from function code during execution.

Type: `Environment` object

Required: No

**EphemeralStorage**

The size of the function's `/tmp` directory in MB. The default value is 512, but can be any whole number between 512 and 10,240 MB. For more information, see Configuring ephemeral storage (console).

Type: `EphemeralStorage` object

Required: No

**FileSystemConfigs**

Connection settings for an Amazon EFS file system.

Type: Array of `FileSystemConfig` objects

Array Members: Maximum number of 1 item.

Required: No

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** – my-function.

CreateFunction
• **Partial ARN** – `123456789012:function:my-function`.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:(aws[a-zA-Z-]*)?::lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]*)+(:($LATEST|[a-zA-Z0-9-_]*)+)`?

Required: Yes

**Handler**

The name of the method within your code that Lambda calls to run your function. Handler is required if the deployment package is a .zip file archive. The format includes the file name. It can also include namespaces and other qualifiers, depending on the runtime. For more information, see [Lambda programming model](#).

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

Required: No

**ImageConfig**

Container image [configuration values](#) that override the values in the container image Dockerfile.

Type: `ImageConfig` object

Required: No

**KMSKeyArn**

The ARN of the AWS Key Management Service (AWS KMS) customer managed key that's used to encrypt your function's [environment variables](#). When [Lambda SnapStart](#) is activated, Lambda also uses this key is to encrypt your function's snapshot. If you deploy your function using a container image, Lambda also uses this key to encrypt your function when it's deployed. Note that this is not the same key that's used to protect your container image in the Amazon Elastic Compute Cloud (Amazon EC2).
Container Registry (Amazon ECR). If you don't provide a customer managed key, Lambda uses a default service key.

Type: String

Pattern: \(\text{arn}:(\text{aws}[a-zA-Z-]*)?:[a-z0-9-.]+::\)()

Required: No

**Layers**

A list of function layers to add to the function's execution environment. Specify each layer by its ARN, including the version.

Type: Array of strings

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: \(\text{arn}:[a-zA-Z0-9-]+:\text{lambda}:\text{a-zA-Z0-9-}+:\text{d\{12\}:layer}:\text{a-zA-Z0-9-}_+\)\([0-9]+\)

Required: No

**LoggingConfig**

The function's Amazon CloudWatch Logs configuration settings.

Type: **LoggingConfig** object

Required: No

**MemorySize**

The amount of memory available to the function at runtime. Increasing the function memory also increases its CPU allocation. The default value is 128 MB. The value can be any multiple of 1 MB.

Type: Integer


Required: No

**PackageType**

The type of deployment package. Set to Image for container image and set to Zip for .zip file archive.
Type: String

Valid Values: Zip | Image

Required: No

Publish

Set to true to publish the first version of the function during creation.

Type: Boolean

Required: No

Role

The Amazon Resource Name (ARN) of the function's execution role.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@-_/]+

Required: Yes

Runtime

The identifier of the function's runtime. Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see Runtime deprecation policy.

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17 | ruby3.2 | python3.11 | nodejs20.x | provided.al2023 | python3.12 | java21

Required: No
SnapStart

The function's SnapStart setting.

Type: SnapStart object

Required: No

Tags

A list of tags to apply to the function.

Type: String to string map

Required: No

Timeout

The amount of time (in seconds) that Lambda allows a function to run before stopping it. The default is 3 seconds. The maximum allowed value is 900 seconds. For more information, see Lambda execution environment.

Type: Integer

Valid Range: Minimum value of 1.

Required: No

TracingConfig

Set Mode to Active to sample and trace a subset of incoming requests with X-Ray.

Type: TracingConfig object

Required: No

VpcConfig

For network connectivity to AWS resources in a VPC, specify a list of security groups and subnets in the VPC. When you connect a function to a VPC, it can access resources and the internet only through that VPC. For more information, see Configuring a Lambda function to access resources in a VPC.

Type: VpcConfig object

Required: No
Response Syntax

HTTP/1.1 201
Content-type: application/json

{
   "Architectures": [ "string" ],
   "CodeSha256": "string",
   "CodeSize": number,
   "DeadLetterConfig": {
      "TargetArn": "string"
   },
   "Description": "string",
   "Environment": {
      "Error": {
         "ErrorCode": "string",
         "Message": "string"
      },
      "Variables": {
         "string": "string"
      }
   },
   "EphemeralStorage": {
      "Size": number
   },
   "FileSystemConfigs": [
      {
         "Arn": "string",
         "LocalMountPath": "string"
      }
   ],
   "FunctionArn": "string",
   "FunctionName": "string",
   "Handler": "string",
   "ImageConfigResponse": {
      "Error": {
         "ErrorCode": "string",
         "Message": "string"
      },
      "ImageConfig": {
         "Command": [ "string" ],
         "EntryPoint": [ "string" ],
         "WorkingDirectory": "string"
      }
   }
}
"KMSKeyArn": "string",
"LastModified": "string",
"LastUpdateStatus": "string",
"LastUpdateStatusReason": "string",
"LastUpdateStatusReasonCode": "string",
"Layers": [
  {
    "Arn": "string",
    "CodeSize": "number",
    "SigningJobArn": "string",
    "SigningProfileVersionArn": "string"
  }
],
"LoggingConfig": {
  "ApplicationLogLevel": "string",
  "LogFormat": "string",
  "LogGroup": "string",
  "SystemLogLevel": "string"
},
"MasterArn": "string",
"MemorySize": "number",
"PackageType": "string",
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"RuntimeVersionConfig": {
  "Error": {
    "ErrorCode": "string",
    "Message": "string"
  },
  "RuntimeVersionArn": "string"
},
"SigningJobArn": "string",
"SigningProfileVersionArn": "string",
"SnapStart": {
  "ApplyOn": "string",
  "OptimizationStatus": "string"
},
"State": "string",
"StateReason": "string",
"StateReasonCode": "string",
"Timeout": "number"
"TracingConfig": {
    "Mode": "string"
},
"Version": "string",
"VpcConfig": {
    "Ipv6AllowedForDualStack": boolean,
    "SecurityGroupIds": [ "string" ],
    "SubnetIds": [ "string" ],
    "VpcId": "string"
}

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

Architectures

The instruction set architecture that the function supports. Architecture is a string array with one of the valid values. The default architecture value is x86_64.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: x86_64 | arm64

CodeSha256

The SHA256 hash of the function's deployment package.

Type: String

CodeSize

The size of the function's deployment package, in bytes.

Type: Long

DeadLetterConfig

The function's dead letter queue.
Type: DeadLetterConfig object

**Description**

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

**Environment**

The function's environment variables. Omitted from AWS CloudTrail logs.

Type: EnvironmentResponse object

**EphemeralStorage**

The size of the function's /tmp directory in MB. The default value is 512, but can be any whole number between 512 and 10,240 MB. For more information, see Configuring ephemeral storage (console).

Type: EphemeralStorage object

**FileSystemConfigs**

Connection settings for an Amazon EFS file system.

Type: Array of FileSystemConfig objects

Array Members: Maximum number of 1 item.

**FunctionArn**

The function's Amazon Resource Name (ARN).

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}: \d{12}:function:[a-zA-Z0-9-_.]+(:($LATEST|[a-zA-Z0-9-_.]+))?

**FunctionName**

The name of the function.

Type: String

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_.]+)(:(\$LATEST|[a-zA-Z0-9-_.]+))?  

**Handler**

The function that Lambda calls to begin running your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: [^\s]+  

**ImageConfigResponse**

The function's image configuration values.

Type: ImageConfigResponse object  

**KMSKeyArn**

The AWS KMS key that's used to encrypt the function's environment variables. When Lambda SnapStart is activated, this key is also used to encrypt the function's snapshot. This key is returned only if you've configured a customer managed key.

Type: String

Pattern: (arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-.]+::)|()  

**LastModified**

The date and time that the function was last updated, in **ISO-8601 format** (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String  

**LastUpdateStatus**

The status of the last update that was performed on the function. This is first set to Successful after function creation completes.

Type: String

Valid Values: Successful | Failed | InProgress
**LastUpdateStatusReason**

The reason for the last update that was performed on the function.

Type: String

**LastUpdateStatusReasonCode**

The reason code for the last update that was performed on the function.

Type: String


**Layers**

The function's layers.

Type: Array of Layer objects

**LoggingConfig**

The function's Amazon CloudWatch Logs configuration settings.

Type: LoggingConfig object

**MasterArn**

For Lambda@Edge functions, the ARN of the main function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}: \d{12}:function:[a-zA-Z0-9-_.]+(:(\$LATEST|[a-zA-Z0-9-_.]+))?

**MemorySize**

The amount of memory available to the function at runtime.
Type: Integer


**PackageType**

The type of deployment package. Set to Image for container image and set Zip for .zip file archive.

Type: String

Valid Values: Zip | Image

**RevisionId**

The latest updated revision of the function or alias.

Type: String

**Role**

The function's execution role.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?::iam::\d{12}:role/\?[a-zA-Z_0-9+=,.@\-_\/]+?

**Runtime**

The identifier of the function's runtime. Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see Runtime deprecation policy.

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17 | ruby3.2 | python3.11 | nodejs20.x | provided.al2023 | python3.12 | java21
**RuntimeVersionConfig**

The ARN of the runtime and any errors that occurred.

Type: `RuntimeVersionConfig` object

**SigningJobArn**

The ARN of the signing job.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9-]+)([a-z]{2}(-gov)?-[a-z]+-\d{1})?:\d{12}?:(.*)`

**SigningProfileVersionArn**

The ARN of the signing profile version.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9-]+)([a-z]{2}(-gov)?-[a-z]+-\d{1})?:\d{12}?:(.*)`

**SnapStart**

Set `ApplyOn` to `PublishedVersions` to create a snapshot of the initialized execution environment when you publish a function version. For more information, see [Improving startup performance with Lambda SnapStart](#).

Type: `SnapStartResponse` object

**State**

The current state of the function. When the state is `Inactive`, you can reactivate the function by invoking it.

Type: String

Valid Values: Pending | Active | Inactive | Failed

**StateReason**

The reason for the function's current state.

Type: String
**StateReasonCode**

The reason code for the function's current state. When the code is Creating, you can't invoke or modify the function.

Type: String


**Timeout**

The amount of time in seconds that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

**TracingConfig**

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse object](#)

**Version**

The version of the Lambda function.

Type: String


Pattern: (\$LATEST|[0-9]+)

**VpcConfig**

The function's networking configuration.

Type: [VpcConfigResponse object](#)
Errors

**CodeSigningConfigNotFoundException**

The specified code signing configuration does not exist.

HTTP Status Code: 404

**CodeStorageExceededException**

Your AWS account has exceeded its maximum total code size. For more information, see Lambda quotas.

HTTP Status Code: 400

**CodeVerificationFailedException**

The code signature failed one or more of the validation checks for signature mismatch or expiry, and the code signing policy is set to ENFORCE. Lambda blocks the deployment.

HTTP Status Code: 400

**InvalidCodeSignatureException**

The code signature failed the integrity check. If the integrity check fails, then Lambda blocks deployment, even if the code signing policy is set to WARN.

HTTP Status Code: 400

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceConflictException**

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404
**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
CreateFunctionUrlConfig

Creates a Lambda function URL with the specified configuration parameters. A function URL is a dedicated HTTP(S) endpoint that you can use to invoke your function.

Request Syntax

```
POST /2021-10-31/functions/FunctionName/url?Qualifier=Qualifier HTTP/1.1
Content-type: application/json

{
  "AuthType": "string",
  "Cors": {
    "AllowCredentials": boolean,
    "AllowHeaders": [ "string" ],
    "AllowMethods": [ "string" ],
    "AllowOrigins": [ "string" ],
    "ExposeHeaders": [ "string" ],
    "MaxAge": number
  },
  "InvokeMode": "string"
}
```

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.
Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-\_]+)(:(\$LATEST|[a-zA-Z0-9-\_]+))?

Required: Yes

**Qualifier**

The alias name.


Pattern: (^\$LATEST$)|((?!^[0-9]+$)([a-zA-Z0-9-\_]+))

**Request Body**

The request accepts the following data in JSON format.

**AuthType**

The type of authentication that your function URL uses. Set to AWS_IAM if you want to restrict access to authenticated users only. Set to NONE if you want to bypass IAM authentication to create a public endpoint. For more information, see Security and auth model for Lambda function URLs.

Type: String

Valid Values: NONE | AWS_IAM

Required: Yes

**Cors**

The cross-origin resource sharing (CORS) settings for your function URL.

Type: Cors object

Required: No

**InvokeMode**

Use one of the following options:

- BUFFERED – This is the default option. Lambda invokes your function using the Invoke API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.
RESPONSE_STREAM – Your function streams payload results as they become available. Lambda invokes your function using the InvokeWithResponseStream API operation. The maximum response payload size is 20 MB, however, you can request a quota increase.

Type: String

Valid Values: BUFFERED | RESPONSE_STREAM

Required: No

Response Syntax

HTTP/1.1 201
Content-type: application/json

{
   "AuthType": "string",
   "Core": {
      "AllowCredentials": boolean,
      "AllowHeaders": [ "string" ],
      "AllowMethods": [ "string" ],
      "AllowOrigins": [ "string" ],
      "ExposeHeaders": [ "string" ],
      "MaxAge": number
   },
   "CreationTime": "string",
   "FunctionArn": "string",
   "FunctionUrl": "string",
   "InvokeMode": "string"
}

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

**AuthType**

The type of authentication that your function URL uses. Set to AWS_IAM if you want to restrict access to authenticated users only. Set to NONE if you want to bypass IAM authentication to
create a public endpoint. For more information, see Security and auth model for Lambda function URLs.

Type: String

Valid Values: NONE | AWS_IAM

**Cors**

The cross-origin resource sharing (CORS) settings for your function URL.

Type: Cors object

**CreationTime**

When the function URL was created, in ISO-8601 format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

**FunctionArn**

The Amazon Resource Name (ARN) of your function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-\_]\+$LATEST|\[a-zA-Z0-9-\_]+\)?

**FunctionUrl**

The HTTP URL endpoint for your function.

Type: String

Length Constraints: Minimum length of 40. Maximum length of 100.

**InvokeMode**

Use one of the following options:

- BUFFERED – This is the default option. Lambda invokes your function using the Invoke API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.
- RESPONSE_STREAM – Your function streams payload results as they become available. Lambda invokes your function using the InvokeWithResponseStream API operation. The maximum response payload size is 20 MB, however, you can request a quota increase.
Type: String

Valid Values: BUFFERED | RESPONSE_STREAM

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
• AWS SDK for C++
• AWS SDK for Go
• AWS SDK for Java V2
• AWS SDK for JavaScript V3
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
DeleteAlias

Deletes a Lambda function alias.

Request Syntax

```
DELETE /2015-03-31/functions/FunctionName/aliases/Name HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** - MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

```
(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-\_]+)(:($LATEST|[a-zA-Z0-9-\_]+))?
```

Required: Yes

**Name**

The name of the alias.


```
(?!^[0-9]+$)([a-zA-Z0-9-\_]+)
```

Required: Yes
Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 204

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:
• **AWS Command Line Interface**
• **AWS SDK for .NET**
• **AWS SDK for C++**
• **AWS SDK for Go**
• **AWS SDK for Java V2**
• **AWS SDK for JavaScript V3**
• **AWS SDK for PHP V3**
• **AWS SDK for Python**
• **AWS SDK for Ruby V3**
DeleteCodeSigningConfig

Deletes the code signing configuration. You can delete the code signing configuration only if no function is using it.

Request Syntax

```
DELETE /2020-04-22/code-signing-configs/CodeSigningConfigArn HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

- **CodeSigningConfigArn**
  
  The The Amazon Resource Name (ARN) of the code signing configuration.
  
  Length Constraints: Maximum length of 200.
  
  Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}((-gov)|(-iso(b?)))?-\d{1}:/\d{12}:code-signing-config:csc-\d{1}7
  
  Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

- **InvalidParameterValueException**
  
  One of the parameters in the request is not valid.
HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
DeleteEventSourceMapping

Deletes an event source mapping. You can get the identifier of a mapping from the output of ListEventSourceMappings.

When you delete an event source mapping, it enters a Deleting state and might not be completely deleted for several seconds.

Request Syntax

DELETE /2015-03-31/event-source-mappings/UUID HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**UUID**

The identifier of the event source mapping.

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 202
Content-type: application/json

```json
{
    "AmazonManagedKafkaEventSourceConfig": {
        "ConsumerGroupId": "string"
    },
    "BatchSize": number,
    "BisectBatchOnFunctionError": boolean,
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
```

"Destination": "string",
},
"DocumentDBEventSourceConfig": {
"CollectionName": "string",
"DatabaseName": "string",
"FullDocument": "string"
},
"EventSourceArn": "string",
"FilterCriteria": {
"Filters": [
  {
    "Pattern": "string"
  }
],
"FunctionArn": "string",
"FunctionResponseTypes": [ "string" ],
"LastModified": number,
"LastProcessingResult": "string",
"MaximumBatchingWindowInSeconds": number,
"MaximumRecordAgeInSeconds": number,
"MaximumRetryAttempts": number,
"ParallelizationFactor": number,
"Queues": [ "string" ],
"ScalingConfig": {
  "MaximumConcurrency": number
},
"SelfManagedEventSource": {
  "Endpoints": {
    "string": [ "string" ]
  }
},
"SelfManagedKafkaEventSourceConfig": {
  "ConsumerGroupId": "string"
},
"SourceAccessConfigurations": [
  {
    "Type": "string",
    "URI": "string"
  }
],
"StartingPosition": "string",
"StartingPositionTimestamp": number,
"State": "string",
"StateTransitionReason": "string",
"Topics": [ "string" ],
"TumblingWindowInSeconds": number,
"UUID": "string"
}

Response Elements

If the action is successful, the service sends back an HTTP 202 response.

The following data is returned in JSON format by the service.

AmazonManagedKafkaEventSourceConfig

Specific configuration settings for an Amazon Managed Streaming for Apache Kafka (Amazon MSK) event source.

Type: AmazonManagedKafkaEventSourceConfig object

BatchSize

The maximum number of records in each batch that Lambda pulls from your stream or queue and sends to your function. Lambda passes all of the records in the batch to the function in a single call, up to the payload limit for synchronous invocation (6 MB).

Default value: Varies by service. For Amazon SQS, the default is 10. For all other services, the default is 100.

Related setting: When you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

BisectBatchOnFunctionError

(Kinesis and DynamoDB Streams only) If the function returns an error, split the batch in two and retry. The default value is false.

Type: Boolean
**DestinationConfig**

(Kinesis and DynamoDB Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: DestinationConfig object

**DocumentDBEventSourceConfig**

Specific configuration settings for a DocumentDB event source.

Type: DocumentDBEventSourceConfig object

**EventSourceArn**

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*)::([a-zA-Z0-9-]+)+([a-zA-Z]{2}(-gov)?-[a-zA-Z]+-\d{1})?:(\d{12})?:(.*)

**FilterCriteria**

An object that defines the filter criteria that determine whether Lambda should process an event. For more information, see [Lambda event filtering](#).

Type: FilterCriteria object

**FunctionArn**

The ARN of the Lambda function.

Type: String

Pattern: arn:(aws[a-zA-Z-A-Z-]*)?:lambda:[a-zA-Z]{2}(-gov)?-[a-zA-Z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-._]+($LATEST|[a-zA-Z0-9-._]+))?

**FunctionResponseTypes**

(Kinesis, DynamoDB Streams, and Amazon SQS) A list of current response type enums applied to the event source mapping.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 1 item.

Valid Values: ReportBatchItemFailures
**LastModified**

The date that the event source mapping was last updated or that its state changed, in Unix time seconds.

Type: Timestamp

**LastProcessingResult**

The result of the last Lambda invocation of your function.

Type: String

**MaximumBatchingWindowInSeconds**

The maximum amount of time, in seconds, that Lambda spends gathering records before invoking the function. You can configure `MaximumBatchingWindowInSeconds` to any value from 0 seconds to 300 seconds in increments of seconds.

For streams and Amazon SQS event sources, the default batching window is 0 seconds. For Amazon MSK, Self-managed Apache Kafka, Amazon MQ, and DocumentDB event sources, the default batching window is 500 ms. Note that because you can only change `MaximumBatchingWindowInSeconds` in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

Related setting: For streams and Amazon SQS event sources, when you set `BatchSize` to a value greater than 10, you must set `MaximumBatchingWindowInSeconds` to at least 1.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

**MaximumRecordAgeInSeconds**

(Kinesis and DynamoDB Streams only) Discard records older than the specified age. The default value is -1, which sets the maximum age to infinite. When the value is set to infinite, Lambda never discards old records.

**Note**

The minimum valid value for maximum record age is 60s. Although values less than 60 and greater than -1 fall within the parameter's absolute range, they are not allowed.
Type: Integer


**MaximumRetryAttempts**

(Kinesis and DynamoDB Streams only) Discard records after the specified number of retries. The default value is -1, which sets the maximum number of retries to infinite. When MaximumRetryAttempts is infinite, Lambda retries failed records until the record expires in the event source.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

**ParallelizationFactor**

(Kinesis and DynamoDB Streams only) The number of batches to process concurrently from each shard. The default value is 1.

Type: Integer


**Queues**

(Amazon MQ) The name of the Amazon MQ broker destination queue to consume.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 1000.

Pattern: [\s\S]*

**ScalingConfig**

(Amazon SQS only) The scaling configuration for the event source. For more information, see Configuring maximum concurrency for Amazon SQS event sources.

Type: **ScalingConfig** object

**SelfManagedEventSource**

The self-managed Apache Kafka cluster for your event source.

Type: **SelfManagedEventSource** object
**SelfManagedKafkaEventSourceConfig**

Specific configuration settings for a self-managed Apache Kafka event source.

Type: `SelfManagedKafkaEventSourceConfig` object

**SourceAccessConfigurations**

An array of the authentication protocol, VPC components, or virtual host to secure and define your event source.

Type: Array of `SourceAccessConfiguration` objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

**StartingPosition**

The position in a stream from which to start reading. Required for Amazon Kinesis and Amazon DynamoDB Stream event sources. AT_TIMESTAMP is supported only for Amazon Kinesis streams, Amazon DocumentDB, Amazon MSK, and self-managed Apache Kafka.

Type: String

Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

**StartingPositionTimestamp**

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds. StartingPositionTimestamp cannot be in the future.

Type: Timestamp

**State**

The state of the event source mapping. It can be one of the following: Creating, Enabling, Enabled, Disabling, Disabled, Updating, or Deleting.

Type: String

**StateTransitionReason**

Indicates whether a user or Lambda made the last change to the event source mapping.

Type: String

**Topics**

The name of the Kafka topic.
Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 249.

Pattern: ^[\^ . ]([a-zA-Z0-9\-_.]+)+

**TumblingWindowInSeconds**

(Kinesis and DynamoDB Streams only) The duration in seconds of a processing window for DynamoDB and Kinesis Streams event sources. A value of 0 seconds indicates no tumbling window.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

**UUID**

The identifier of the event source mapping.

Type: String

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceConflictException**

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

**ResourceInUseException**

The operation conflicts with the resource's availability. For example, you tried to update an event source mapping in the CREATING state, or you tried to delete an event source mapping currently UPDATING.

HTTP Status Code: 400
ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
DeleteFunction

Deletes a Lambda function. To delete a specific function version, use the **Qualifier** parameter. Otherwise, all versions and aliases are deleted. This doesn't require the user to have explicit permissions for **DeleteAlias**.

To delete Lambda event source mappings that invoke a function, use **DeleteEventSourceMapping**. For AWS services and resources that invoke your function directly, delete the trigger in the service where you originally configured it.

Request Syntax

```
DELETE /2015-03-31/functions/FunctionName?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function or version.

**Name formats**

- **Function name** – `my-function` (name-only), `my-function:1` (with version).
- **Partial ARN** – `123456789012:function:my-function`.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: 

```
(arn:(aws[a-zA-Z-]*)?:lambda:)?([-a-zA-Z]{2}([-gov][-a-zA-Z]+-\d{1}:)?\d{12}:)?(function:)?([-a-zA-Z0-9-._]+)(\$LATEST|[a-zA-Z0-9-._]+))?
```

Required: Yes
**Qualifier**

Specify a version to delete. You can't delete a version that an alias references.


Pattern: ( | [a-zA-Z0-9$_-]+ )

**Request Body**

The request does not have a request body.

**Response Syntax**

HTTP/1.1 204

**Response Elements**

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceConflictException**

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.
HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
DeleteFunctionCodeSigningConfig

Removes the code signing configuration from the function.

Request Syntax

DELETE /2020-06-30/functions/FunctionName/code-signing-config HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

FunctionName

The name of the Lambda function.

Name formats

- Function name - MyFunction.
- Partial ARN - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?(\d{12}:)?(function:)?([a-zA-Z0-9-\_]+)(:(\$LATEST|[a-zA-Z0-9-\_]+))?

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 204
Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

**CodeSigningConfigNotFoundException**

The specified code signing configuration does not exist.

HTTP Status Code: 404

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceConflictException**

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:
- **AWS Command Line Interface**
- **AWS SDK for .NET**
- **AWS SDK for C++**
- **AWS SDK for Go**
- **AWS SDK for Java V2**
- **AWS SDK for JavaScript V3**
- **AWS SDK for PHP V3**
- **AWS SDK for Python**
- **AWS SDK for Ruby V3**
DeleteFunctionConcurrency

Removes a concurrent execution limit from a function.

Request Syntax

DELETE /2017-10-31/functions/FunctionName/concurrency HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

FunctionName

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?(\[a-zA-Z0-9-\_]*)\+\+?(:\$LATEST[\[a-zA-Z0-9-\_]+])?

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 204
Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
• AWS SDK for Go
• AWS SDK for Java V2
• AWS SDK for JavaScript V3
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
DeleteFunctionEventInvokeConfig

Deletes the configuration for asynchronous invocation for a function, version, or alias.

To configure options for asynchronous invocation, use [PutFunctionEventInvokeConfig](#).

**Request Syntax**

```
DELETE /2019-09-25/functions/FunctionName/event-invoke-config?Qualifier=Qualifier
HTTP/1.1
```

**URI Request Parameters**

The request uses the following URI parameters.

- **FunctionName**
  - The name of the Lambda function, version, or alias.
  - **Name formats**
    - **Function name** - my-function (name-only), my-function:v1 (with alias).
    - **Partial ARN** - 123456789012:function:my-function.

    You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

    **Length Constraints:** Minimum length of 1. Maximum length of 140.

    **Pattern:** (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-]+)(:(\$LATEST|[a-zA-Z0-9-]+))?

    **Required:** Yes

- **Qualifier**
  - A version number or alias name.

  **Length Constraints:** Minimum length of 1. Maximum length of 128.
Pattern: ( | [a-zA-Z0-9$-_]+ )

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 204

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.
HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
DeleteFunctionUrlConfig

Deletes a Lambda function URL. When you delete a function URL, you can't recover it. Creating a new function URL results in a different URL address.

Request Syntax

DELETE /2021-10-31/functions/{FunctionName}/url?Qualifier={Qualifier} HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

FunctionName

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-[a-z]{12}:)?(function:)?([a-zA-Z0-9-._]+)(:\$LATEST|[a-zA-Z0-9-._]+))?

Required: Yes

Qualifier

The alias name.


Pattern: (^\$LATEST$)|((?!^[0-9]+$)([a-zA-Z0-9-._]+))
Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 204

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:
- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
DeleteLayerVersion

Deletes a version of an AWS Lambda layer. Deleted versions can no longer be viewed or added to functions. To avoid breaking functions, a copy of the version remains in Lambda until no functions refer to it.

Request Syntax

DELETE /2018-10-31/layers/\{LayerName\}/versions/\{VersionNumber\} HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**LayerName**

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+::\d{12}:layer:[a-zA-Z0-9-\_]+)|[a-zA-Z0-9-\_]+

Required: Yes

**VersionNumber**

The version number.

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 204

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.
Errors

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
DeleteProvisionedConcurrencyConfig

Deletes the provisioned concurrency configuration for a function.

Request Syntax

DELETE /2019-09-30/functions/FunctionName/provisioned-concurrency?Qualifier=Qualifier HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

**Length Constraints:** Minimum length of 1. Maximum length of 140.

**Pattern:** (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-]*)((\$(LATEST|[a-zA-Z0-9-]+))?)

Required: Yes

**Qualifier**

The version number or alias name.

**Length Constraints:** Minimum length of 1. Maximum length of 128.

**Pattern:** ([a-zA-Z0-9-]*)_+}
Required: Yes

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 204

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.
HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
GetAccountSettings

Retrieves details about your account's limits and usage in an AWS Region.

Request Syntax

GET /2016-08-19/account-settings/ HTTP/1.1

URI Request Parameters

The request does not use any URI parameters.

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

{
    "AccountLimit": {
        "CodeSizeUnzipped": number,
        "CodeSizeZipped": number,
        "ConcurrentExecutions": number,
        "TotalCodeSize": number,
        "UnreservedConcurrentExecutions": number
    },
    "AccountUsage": {
        "FunctionCount": number,
        "TotalCodeSize": number
    }
}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.
**AccountLimit**

Limits that are related to concurrency and code storage.

Type: `AccountLimit` object

**AccountUsage**

The number of functions and amount of storage in use.

Type: `AccountUsage` object

**Errors**

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
GetAlias

Returns details about a Lambda function alias.

Request Syntax

GET /2015-03-31/functions/FunctionName/aliases/Name HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

FunctionName

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-d{1}:)?(d{12}:)?(function:)?([a-zA-Z0-9-]+)(:($LATEST|[a-zA-Z0-9-]+))?

Required: Yes

Name

The name of the alias.


Pattern: (?!^[0-9]+$)([a-zA-Z0-9-]+)

Required: Yes
Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

{  
    "AliasArn": "string",
    "Description": "string",
    "FunctionVersion": "string",
    "Name": "string",
    "RevisionId": "string",
    "RoutingConfig": {
        "AdditionalVersionWeights": {
            "string": number
        }
    }
}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**AliasArn**

The Amazon Resource Name (ARN) of the alias.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_.]+:(\$LATEST|[a-zA-Z0-9-_.]+)?

**Description**

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.
**FunctionVersion**

The function version that the alias invokes.

Type: String


Pattern: \($LATEST \|[0-9]+\)$

**Name**

The name of the alias.

Type: String


Pattern: (?!^[0-9]+$)([a-zA-Z0-9-]+)

**RevisionId**

A unique identifier that changes when you update the alias.

Type: String

**RoutingConfig**

The routing configuration of the alias.

Type: `AliasRoutingConfiguration` object

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404
ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
**GetCodeSigningConfig**

Returns information about the specified code signing configuration.

**Request Syntax**

```
GET /2020-04-22/code-signing-configs/CodeSigningConfigArn HTTP/1.1
```

**URI Request Parameters**

The request uses the following URI parameters.

**CodeSigningConfigArn**

The The Amazon Resource Name (ARN) of the code signing configuration.

Length Constraints: Maximum length of 200.

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}((-gov)|(-iso(b?)))?-[a-z]+\d{1}:\d{12}:code-signing-config:csc-[a-z0-9]{17}`

Required: Yes

**Request Body**

The request does not have a request body.

**Response Syntax**

```
HTTP/1.1 200
Content-type: application/json

{
    "CodeSigningConfig": {
        "AllowedPublishers": {
            "SigningProfileVersionArns": [ "string" ]
        },
        "CodeSigningConfigArn": "string",
        "CodeSigningConfigId": "string",
        "CodeSigningPolicies": {
            "UntrustedArtifactOnDeployment": "string"
        }
    }
}
```
Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**CodeSigningConfig**

The code signing configuration

Type: CodeSigningConfig object

Errors

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:
• AWS Command Line Interface
• AWS SDK for .NET
• AWS SDK for C++
• AWS SDK for Go
• AWS SDK for Java V2
• AWS SDK for JavaScript V3
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
GetEventSourceMapping

Returns details about an event source mapping. You can get the identifier of a mapping from the output of ListEventSourceMappings.

Request Syntax

GET /2015-03-31/event-source-mappings/UUID HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**UUID**

The identifier of the event source mapping.

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

```
{
    "AmazonManagedKafkaEventSourceConfig": {
        "ConsumerGroupId": "string"
    },
    "BatchSize": number,
    "BisectBatchOnFunctionError": boolean,
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    }
}
```
"DocumentDBEventSourceConfig": {
    "CollectionName": "string",
    "DatabaseName": "string",
    "FullDocument": "string"
},
"EventSourceArn": "string",
"FilterCriteria": {
    "Filters": [
    {
        "Pattern": "string"
    }
]
},
"FunctionArn": "string",
"FunctionResponseTypes": [ "string" ],
"LastModified": number,
"LastProcessingResult": "string",
"MaximumBatchingWindowInSeconds": number,
"MaximumRecordAgeInSeconds": number,
"MaximumRetryAttempts": number,
"ParallelizationFactor": number,
"Queues": [ "string" ],
"ScalingConfig": {
    "MaximumConcurrency": number
},
"SelfManagedEventSource": {
    "Endpoints": {
        "string": [ "string" ]
    }
},
"SelfManagedKafkaEventSourceConfig": {
    "ConsumerGroupId": "string"
},
"SourceAccessConfigurations": [
    {
        "Type": "string",
        "URI": "string"
    }
],
"StartingPosition": "string",
"StartingPositionTimestamp": number,
"State": "string",
"StateTransitionReason": "string",
"Topics": [ "string" ],
"TumblingWindowInSeconds": number,
"UUID": "string"
}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**AmazonManagedKafkaEventSourceConfig**

Specific configuration settings for an Amazon Managed Streaming for Apache Kafka (Amazon MSK) event source.

Type: AmazonManagedKafkaEventSourceConfig object

**BatchSize**

The maximum number of records in each batch that Lambda pulls from your stream or queue and sends to your function. Lambda passes all of the records in the batch to the function in a single call, up to the payload limit for synchronous invocation (6 MB).

Default value: Varies by service. For Amazon SQS, the default is 10. For all other services, the default is 100.

Related setting: When you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowIn Seconds to at least 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

**BisectBatchOnFunctionError**

(Kinesis and DynamoDB Streams only) If the function returns an error, split the batch in two and retry. The default value is false.

Type: Boolean

**DestinationConfig**

(Kinesis and DynamoDB Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.
Type: DestinationConfig object

**DocumentDBEventSourceConfig**

Specific configuration settings for a DocumentDB event source.

Type: DocumentDBEventSourceConfig object

**EventSourceArn**

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9-]+)+:([a-z]{2}(-gov)?-[a-z]+-[\d\{1}\{12}\{.*})`

**FilterCriteria**

An object that defines the filter criteria that determine whether Lambda should process an event. For more information, see Lambda event filtering.

Type: FilterCriteria object

**FunctionArn**

The ARN of the Lambda function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-zA-Z0-9-]+(-gov)?-[a-z]+-\d\{1\}:\d\{12\}:function:[a-zA-Z0-9-]+(:\($LATEST|[a-zA-Z0-9-]+\))?

**FunctionResponseTypes**

(Kinesis, DynamoDB Streams, and Amazon SQS) A list of current response type enums applied to the event source mapping.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 1 item.

Valid Values: ReportBatchItemFailures

**LastModified**

The date that the event source mapping was last updated or that its state changed, in Unix time seconds.
Last Processing Result

The result of the last Lambda invocation of your function.

Type: String

Maximum Batching Window In Seconds

The maximum amount of time, in seconds, that Lambda spends gathering records before invoking the function. You can configure Maximum Batching Window In Seconds to any value from 0 seconds to 300 seconds in increments of seconds.

For streams and Amazon SQS event sources, the default batching window is 0 seconds. For Amazon MSK, Self-managed Apache Kafka, Amazon MQ, and DocumentDB event sources, the default batching window is 500 ms. Note that because you can only change Maximum Batching Window In Seconds in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

Related setting: For streams and Amazon SQS event sources, when you set Batch Size to a value greater than 10, you must set Maximum Batching Window In Seconds to at least 1.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

Maximum Record Age In Seconds

(Kinesis and DynamoDB Streams only) Discard records older than the specified age. The default value is -1, which sets the maximum age to infinite. When the value is set to infinite, Lambda never discards old records.

Note

The minimum valid value for maximum record age is 60s. Although values less than 60 and greater than -1 fall within the parameter's absolute range, they are not allowed

Type: Integer

**MaximumRetryAttempts**

(Kinesis and DynamoDB Streams only) Discard records after the specified number of retries. The default value is -1, which sets the maximum number of retries to infinite. When MaximumRetryAttempts is infinite, Lambda retries failed records until the record expires in the event source.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

**ParallelizationFactor**

(Kinesis and DynamoDB Streams only) The number of batches to process concurrently from each shard. The default value is 1.

Type: Integer


**Queues**

(Amazon MQ) The name of the Amazon MQ broker destination queue to consume.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 1000.

Pattern: [\s\S]*

**ScalingConfig**

(Amazon SQS only) The scaling configuration for the event source. For more information, see Configuring maximum concurrency for Amazon SQS event sources.

Type: ScalingConfig object

**SelfManagedEventSource**

The self-managed Apache Kafka cluster for your event source.

Type: SelfManagedEventSource object
SelfManagedKafkaEventSourceConfig

Specific configuration settings for a self-managed Apache Kafka event source.

Type: SelfManagedKafkaEventSourceConfig object

SourceAccessConfigurations

An array of the authentication protocol, VPC components, or virtual host to secure and define your event source.

Type: Array of SourceAccessConfiguration objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

StartingPosition

The position in a stream from which to start reading. Required for Amazon Kinesis and Amazon DynamoDB Stream event sources. AT_TIMESTAMP is supported only for Amazon Kinesis streams, Amazon DocumentDB, Amazon MSK, and self-managed Apache Kafka.

Type: String

Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

StartingPositionTimestamp

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds. StartingPositionTimestamp cannot be in the future.

Type: Timestamp

State

The state of the event source mapping. It can be one of the following: Creating, Enabling, Enabled, Disabling, Disabled, Updating, or Deleting.

Type: String

StateTransitionReason

Indicates whether a user or Lambda made the last change to the event source mapping.

Type: String
Topics

The name of the Kafka topic.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 249.

Pattern: ^[^.](\[a-zA-Z0-9\-_.\]+)

TumblingWindowInSeconds

(Kinesis and DynamoDB Streams only) The duration in seconds of a processing window for DynamoDB and Kinesis Streams event sources. A value of 0 seconds indicates no tumbling window.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

UUID

The identifier of the event source mapping.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.
HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
GetFunction

Returns information about the function or function version, with a link to download the deployment package that's valid for 10 minutes. If you specify a function version, only details that are specific to that version are returned.

Request Syntax

GET /2015-03-31/functions/FunctionName?Qualifier=Qualifier HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function, version, or alias.

**Name formats**

- **Function name** – my-function (name-only), my-function:v1 (with alias).
- **Partial ARN** – 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.


Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)? (\d{12}:)?(function:)?([a-zA-Z0-9-_.]+)(:(\$LATEST|[a-zA-Z0-9-_.]+))? Required: Yes

**Qualifier**

Specify a version or alias to get details about a published version of the function.


Pattern: ([a-zA-Z0-9$-_]+)
Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

{
   "Code": {
      "ImageUri": "string",
      "Location": "string",
      "RepositoryType": "string",
      "ResolvedImageUri": "string"
   },
   "Concurrency": {
      "ReservedConcurrentExecutions": number
   },
   "Configuration": {
      "Architectures": [ "string" ],
      "CodeSha256": "string",
      "CodeSize": number,
      "DeadLetterConfig": {
         "TargetArn": "string"
      },
      "Description": "string",
      "Environment": {
         "Error": {
            "ErrorCode": "string",
            "Message": "string"
         },
         "Variables": {
            "string": "string"
         }
      },
      "EphemeralStorage": {
         "Size": number
      },
      "FileSystemConfigs": [
         {
            "Arn": "string",
            "LocalMountPath": "string"
         }
      ]
   }
}
"FunctionArn": "string",
"FunctionName": "string",
"Handler": "string",
"ImageConfigResponse": {
  "Error": {
    "ErrorCode": "string",
    "Message": "string"
  },
  "ImageConfig": {
    "Command": [ "string" ],
    "EntryPoint": [ "string" ],
    "WorkingDirectory": "string"
  }
},
"KMSKeyArn": "string",
"LastModified": "string",
"LastUpdateStatus": "string",
"LastUpdateStatusReason": "string",
"LastUpdateStatusReasonCode": "string",
"Layers": [
  {
    "Arn": "string",
    "CodeSize": number,
    "SigningJobArn": "string",
    "SigningProfileVersionArn": "string"
  }
],
"LoggingConfig": {
  "ApplicationLogLevel": "string",
  "LogFormat": "string",
  "LogGroup": "string",
  "SystemLogLevel": "string"
},
"MasterArn": "string",
"MemorySize": number,
"PackageType": "string",
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"RuntimeVersionConfig": {
  "Error": {
    "ErrorCode": "string",
    "Message": "string"
  },
  "ImageConfig": {
    "Command": [ "string" ],
    "EntryPoint": [ "string" ],
    "WorkingDirectory": "string"
  }
}
Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**Code**

The deployment package of the function or version.

Type: `FunctionCodeLocation` object

**Concurrency**

The function's `reserved concurrency`.
Type: **Concurrency** object

**Configuration**

The configuration of the function or version.

Type: **FunctionConfiguration** object

**Tags**

The function's **tags**.

Type: String to string map

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:
- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
GetFunctionCodeSigningConfig

Returns the code signing configuration for the specified function.

Request Syntax

GET /2020-06-30/functions/FunctionName/code-signing-config HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** - MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

**Length Constraints:** Minimum length of 1. Maximum length of 140.

**Pattern:** (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-].*+):($LATEST|[a-zA-Z0-9-].*+))?

**Required:** Yes

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json
Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**CodeSigningConfigArn**

The The Amazon Resource Name (ARN) of the code signing configuration.

Type: String

Length Constraints: Maximum length of 200.

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}((-gov)|(-iso(b?)))-[a-z]+-\d{1}:-\d{12}:code-signing-config:csc-[a-z0-9]{17}

**FunctionName**

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]*)?($LATEST|[a-zA-Z0-9-_]*)?\{[a-z\d\-]+\}?
Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
GetFunctionConcurrency

Returns details about the reserved concurrency configuration for a function. To set a concurrency limit for a function, use PutFunctionConcurrency.

Request Syntax

```
GET /2019-09-30/functions/FunctionName/concurrency HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-._]+)(:($LATEST|\d{12}:))?

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
```
Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

ReservedConcurrentExecutions

The number of simultaneous executions that are reserved for the function.

Type: Integer

Valid Range: Minimum value of 0.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429
For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
GetFunctionConfiguration

Returns the version-specific settings of a Lambda function or version. The output includes only options that can vary between versions of a function. To modify these settings, use UpdateFunctionConfiguration.

To get all of a function's details, including function-level settings, use GetFunction.

Request Syntax

GET /2015-03-31/functions/FunctionName/configuration?Qualifier=Qualifier HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function, version, or alias.

**Name formats**

- **Function name** — my-function (name-only), my-function:v1 (with alias).
- **Partial ARN** — 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.


Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}):(\d{12}):(function:)?([a-zA-Z0-9-._]+)(:($LATEST|[a-zA-Z0-9-._]+))?  

Required: Yes

**Qualifier**

Specify a version or alias to get details about a published version of the function.

Pattern: (\[[a-zA-Z0-9$_-]+\])

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

{
   "Architectures": [ "string" ],
   "CodeSha256": "string",
   "CodeSize": number,
   "DeadLetterConfig": {
      "TargetArn": "string"
   },
   "Description": "string",
   "Environment": {
      "Error": {
         "ErrorCode": "string",
         "Message": "string"
      },
      "Variables": {
         "string": "string"
      }
   },
   "EphemeralStorage": {
      "Size": number
   },
   "FileSystemConfigs": [
      {
         "Arn": "string",
         "LocalMountPath": "string"
      }
   ],
   "FunctionArn": "string",
   "FunctionName": "string",
   "Handler": "string",
   "ImageConfigResponse": {
      "Error": {
         "ErrorCode": "string"
      }
   }
}
"Message": "string",
],
"ImageConfig": {
    "Command": [ "string" ],
    "EntryPoint": [ "string" ],
    "WorkingDirectory": "string"
}
],
"KMSKeyArn": "string",
"LastModified": "string",
"LastUpdateStatus": "string",
"LastUpdateStatusReason": "string",
"LastUpdateStatusReasonCode": "string",
"Layers": [
    {
        "Arn": "string",
        "CodeSize": number,
        "SigningJobArn": "string",
        "SigningProfileVersionArn": "string"
    }
],
"LoggingConfig": {
    "ApplicationLogLevel": "string",
    "LogFormat": "string",
    "LogGroup": "string",
    "SystemLogLevel": "string"
},
"MasterArn": "string",
"MemorySize": number,
"PackageType": "string",
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"RuntimeVersionConfig": {
    "Error": {
        "ErrorCode": "string",
        "Message": "string"
    },
    "RuntimeVersionArn": "string"
},
"SigningJobArn": "string",
"SigningProfileVersionArn": "string",
"SnapStart": {
    "ApplyOn": "string"
}
Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Architectures

The instruction set architecture that the function supports. Architecture is a string array with one of the valid values. The default architecture value is `x86_64`.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: `x86_64` | `arm64`

CodeSha256

The SHA256 hash of the function's deployment package.

Type: String

CodeSize

The size of the function's deployment package, in bytes.
DeadLetterConfig

The function's dead letter queue.

Type: DeadLetterConfig object

Description

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Environment

The function's environment variables. Omitted from AWS CloudTrail logs.

Type: EnvironmentResponse object

EphemeralStorage

The size of the function's /tmp directory in MB. The default value is 512, but can be any whole number between 512 and 10,240 MB. For more information, see Configuring ephemeral storage (console).

Type: EphemeralStorage object

FileSystemConfigs

Connection settings for an Amazon EFS file system.

Type: Array of FileSystemConfig objects

Array Members: Maximum number of 1 item.

FunctionArn

The function's Amazon Resource Name (ARN).

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_.]+(:($LATEST|[a-zA-Z0-9-_.]+)?)
**FunctionName**

The name of the function.

Type: String


Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?(\[a-z]{2}(-gov)?-\[a-zA-Z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-._]+)(:\($LATEST|[a-zA-Z0-9-._]+$\))?

**Handler**

The function that Lambda calls to begin running your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: [^\s]+

**ImageConfigResponse**

The function's image configuration values.

Type: ImageConfigResponse object

**KMSKeyArn**

The AWS KMS key that's used to encrypt the function's environment variables. When Lambda SnapStart is activated, this key is also used to encrypt the function's snapshot. This key is returned only if you've configured a customer managed key.

Type: String

Pattern: (arn:(aws[a-zA-Z-]*)?:\[a-z0-9-._]+:.*)\(|()?

**LastModified**

The date and time that the function was last updated, in ISO-8601 format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

**LastUpdateStatus**

The status of the last update that was performed on the function. This is first set to Successful after function creation completes.
**Type:** String

**Valid Values:** Successful | Failed | InProgress

**LastUpdateStatusReason**

The reason for the last update that was performed on the function.

**Type:** String

**LastUpdateStatusReasonCode**

The reason code for the last update that was performed on the function.

**Type:** String

**Valid Values:**
- EniLimitExceeded
- InsufficientRolePermissions
- InvalidConfiguration
- InternalError
- SubnetOutOfIPAddresses
- InvalidSubnet
- InvalidSecurityGroup
- ImageDeleted
- ImageAccessDenied
- InvalidImage
- KMSKeyAccessDenied
- KMSKeyNotFound
- InvalidStateKMSKey
- DisabledKMSKey
- EFSIOError
- EFSMountConnectivityError
- EFSMountFailure
- EFSMountTimeout
- InvalidRuntime
- InvalidZipFileException
- FunctionError

**Layers**

The function's layers.

**Type:** Array of **Layer** objects

**LoggingConfig**

The function's Amazon CloudWatch Logs configuration settings.

**Type:** **LoggingConfig** object

**MasterArn**

For Lambda@Edge functions, the ARN of the main function.

**Type:** String

**Pattern:** arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_.]+\:(\$LATEST|[a-zA-Z0-9-_.]+)?
**MemorySize**

The amount of memory available to the function at runtime.

Type: Integer


**PackageType**

The type of deployment package. Set to Image for container image and set Zip for .zip file archive.

Type: String

Valid Values: Zip | Image

**RevisionId**

The latest updated revision of the function or alias.

Type: String

**Role**

The function's execution role.

Type: String

Pattern: arn:(aws[a-zA-Z-]*):iam::\d{12}:role/?[a-zA-Z_0-9+=,.@/-]+

**Runtime**

The identifier of the function's runtime. Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see Runtime deprecation policy.

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 |
dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6
| nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2
| nodejs18.x | python3.10 | java17 | ruby3.2 | python3.11 | nodejs20.x | provided.al2023 | python3.12 | java21

**RuntimeVersionConfig**

The ARN of the runtime and any errors that occurred.

Type: [RuntimeVersionConfig](#) object

**SigningJobArn**

The ARN of the signing job.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9-]+):([a-zA-Z0-9\d{}\[\]\.,]*)?:.*`

**SigningProfileVersionArn**

The ARN of the signing profile version.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9-]+):([a-zA-Z0-9\d{}\[\]\.,]*)?:.*`

**SnapStart**

Set `ApplyOn` to `PublishedVersions` to create a snapshot of the initialized execution environment when you publish a function version. For more information, see [Improving startup performance with Lambda SnapStart](#).

Type: [SnapStartResponse](#) object

**State**

The current state of the function. When the state is `Inactive`, you can reactivate the function by invoking it.

Type: String

Valid Values: Pending | Active | Inactive | Failed
**StateReason**

The reason for the function's current state.

Type: String

**StateReasonCode**

The reason code for the function's current state. When the code is Creating, you can't invoke or modify the function.

Type: String


**Timeout**

The amount of time in seconds that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

**TracingConfig**

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse object](#)

**Version**

The version of the Lambda function.

Type: String


Pattern: (\$LATEST|[0-9]+)
VpcConfig

The function's networking configuration.

Type: VpcConfigResponse object

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
• AWS SDK for JavaScript V3
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
GetFunctionEventInvokeConfig

Retrieves the configuration for asynchronous invocation for a function, version, or alias.

To configure options for asynchronous invocation, use GetFunctionEventInvokeConfig.

Request Syntax

GET /2019-09-25/functions/FunctionName/event-invoke-config?Qualifier=Qualifier HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function, version, or alias.

**Name formats**

- **Function name** - my-function (name-only), my-function:v1 (with alias).
- **Partial ARN** - 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

**Length Constraints:** Minimum length of 1. Maximum length of 140.

**Pattern:** (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)? (\d{12}:)?(function:)?([a-zA-Z0-9-._]+)(:($LATEST|[a-zA-Z0-9-._]+))? (\d{13}:)?

**Required:** Yes

**Qualifier**

A version number or alias name.

**Length Constraints:** Minimum length of 1. Maximum length of 128.

**Pattern:** ([a-zA-Z0-9-9$-]+)
Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "FunctionArn": "string",
    "LastModified": number,
    "MaximumEventAgeInSeconds": number,
    "MaximumRetryAttempts": number
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**DestinationConfig**

A destination for events after they have been sent to a function for processing.

**Destinations**

- **Function** - The Amazon Resource Name (ARN) of a Lambda function.
- **Queue** - The ARN of a standard SQS queue.
- **Topic** - The ARN of a standard SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: **DestinationConfig** object
**FunctionArn**

The Amazon Resource Name (ARN) of the function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:($LATEST|[a-zA-Z0-9-_]+))?`

**LastModified**

The date and time that the configuration was last updated, in Unix time seconds.

Type: Timestamp

**MaximumEventAgeInSeconds**

The maximum age of a request that Lambda sends to a function for processing.

Type: Integer

Valid Range: Minimum value of 60. Maximum value of 21600.

**MaximumRetryAttempts**

The maximum number of times to retry when the function returns an error.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 2.

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404
**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
GetFunctionUrlConfig

Returns details about a Lambda function URL.

**Request Syntax**

```
GET /2021-10-31/functions/FunctionName/url?Qualifier=Qualifier HTTP/1.1
```

**URI Request Parameters**

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** — my-function.
- **Partial ARN** — 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

```
Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-\_]+)(:\{$LATEST|[a-zA-Z0-9-\_]+\})?
```

Required: Yes

**Qualifier**

The alias name.


```
Pattern: (^\$LATEST$)|((?!^[0-9]+$)([a-zA-Z0-9-\_]+))
```
Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

{
   "AuthType": "string",
   "Cors": {
      "AllowCredentials": boolean,
      "AllowHeaders": [ "string" ],
      "AllowMethods": [ "string" ],
      "AllowOrigins": [ "string" ],
      "ExposeHeaders": [ "string" ],
      "MaxAge": number
   },
   "CreationTime": "string",
   "FunctionArn": "string",
   "FunctionUrl": "string",
   "InvokeMode": "string",
   "LastModifiedTime": "string"
}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**AuthType**

The type of authentication that your function URL uses. Set to AWS_IAM if you want to restrict access to authenticated users only. Set to NONE if you want to bypass IAM authentication to create a public endpoint. For more information, see Security and auth model for Lambda function URLs.

Type: String

Valid Values: NONE | AWS_IAM
Cors

The cross-origin resource sharing (CORS) settings for your function URL.

Type: Cors object

CreationTime

When the function URL was created, in ISO-8601 format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

FunctionArn

The Amazon Resource Name (ARN) of your function.

Type: String

Pattern: arn:(aws[\a-zA-Z-\]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:
\d{12}:function:[a-zA-Z0-9-\_]+(:(\$LATEST|[a-zA-Z0-9-\_]+))?}

FunctionUrl

The HTTP URL endpoint for your function.

Type: String

Length Constraints: Minimum length of 40. Maximum length of 100.

InvokeMode

Use one of the following options:

- BUFFERED – This is the default option. Lambda invokes your function using the Invoke API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.
- RESPONSE_STREAM – Your function streams payload results as they become available. Lambda invokes your function using the InvokeWithResponseStream API operation. The maximum response payload size is 20 MB, however, you can request a quota increase.

Type: String

Valid Values: BUFFERED | RESPONSE_STREAM
LastModifiedTime

When the function URL configuration was last updated, in ISO-8601 format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
GetLayerVersion

Returns information about a version of an AWS Lambda layer, with a link to download the layer archive that's valid for 10 minutes.

Request Syntax

GET /2018-10-31/layers/LayerName/versions/VersionNumber HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

LayerName

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:([a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:):\d{12}:layer:[a-zA-Z0-9-]+)\|[a-zA-Z0-9-]+

Required: Yes

VersionNumber

The version number.

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

{  
  "CompatibleArchitectures": [ "string" ],
  "CompatibleRuntimes": [ "string" ],
}
"Content": {
"CodeSha256": "string",
"CodeSize": number,
"Location": "string",
"SigningJobArn": "string",
"SigningProfileVersionArn": "string"
},
"CreatedDate": "string",
"Description": "string",
"LayerArn": "string",
"LayerVersionArn": "string",
"LicenseInfo": "string",
"Version": number
}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**CompatibleArchitectures**

A list of compatible instruction set architectures.

Type: Array of strings

Array Members: Maximum number of 2 items.

Valid Values: x86_64 | arm64

**CompatibleRuntimes**

The layer's compatible runtimes.

The following list includes deprecated runtimes. For more information, see Runtime deprecation policy.

Type: Array of strings

Array Members: Maximum number of 15 items.

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11
Content

Details about the layer version.

Type: \texttt{LayerVersionContentOutput} object

**CreatedDate**

The date that the layer version was created, in \texttt{ISO-8601} format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

**Description**

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

**LayerArn**

The ARN of the layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: \texttt{arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-\_]+}

**LayerVersionArn**

The ARN of the layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: \texttt{arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-\_]+:\d+[0-9]+}
LicenseInfo

The layer's software license.

Type: String

Length Constraints: Maximum length of 512.

Version

The version number.

Type: Long

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:
- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
GetLayerVersionByArn

Returns information about a version of an AWS Lambda layer, with a link to download the layer archive that's valid for 10 minutes.

Request Syntax

GET /2018-10-31/layers?find=LayerVersion&Arn=Arn HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

Arn

The ARN of the layer version.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-\_]++:[0-9]+

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

{
   "CompatibleArchitectures": [ "string" ],
   "CompatibleRuntimes": [ "string" ],
   "Content": {
      "CodeSha256": "string",
      "CodeSize": number,
      "Location": "string",
   }
}
"SigningJobArn": "string",
"SigningProfileVersionArn": "string"
},

"CreateDate": "string",
"Description": "string",
"LayerArn": "string",
"LayerVersionArn": "string",
"LicenseInfo": "string",

"Version": number
}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**CompatibleArchitectures**

A list of compatible instruction set architectures.

Type: Array of strings

Array Members: Maximum number of 2 items.

Valid Values: x86_64 | arm64

**CompatibleRuntimes**

The layer's compatible runtimes.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: Array of strings

Array Members: Maximum number of 15 items.

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2
Content

Details about the layer version.

Type: LayerVersionContentOutput object

CreatedDate

The date that the layer version was created, in ISO-8601 format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Description

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

LayerArn

The ARN of the layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+://d{12}:layer:[a-zA-Z0-9-__]+:

LayerVersionArn

The ARN of the layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+://d{12}:layer:[a-zA-Z0-9-__]+:[0-9]+:

LicenseInfo

The layer's software license.
Type: String

Length Constraints: Maximum length of 512.

Version

The version number.

Type: Long

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
• AWS SDK for C++
• AWS SDK for Go
• AWS SDK for Java V2
• AWS SDK for JavaScript V3
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
GetLayerVersionPolicy

Returns the permission policy for a version of an AWS Lambda layer. For more information, see AddLayerVersionPermission.

Request Syntax

GET /2018-10-31/layers/LayerName/versions/VersionNumber/policy HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**LayerName**

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-\-_]+)|[a-zA-Z0-9-\-_]+

Required: Yes

**VersionNumber**

The version number.

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

{
   "Policy": "string",
}
Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**Policy**

The policy document.

Type: String

**RevisionId**

A unique identifier for the current revision of the policy.

Type: String

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

** ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).
HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
GetPolicy

Returns the resource-based IAM policy for a function, version, or alias.

Request Syntax

GET /2015-03-31/functions/FunctionName/policy?Qualifier=Qualifier HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function, version, or alias.

**Name formats**

- **Function name** – my-function (name-only), my-function:v1 (with alias).
- **Partial ARN** – 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.


Pattern: (arn:\[(a-zA-Z\-\*)?:lambda?:([a-zA-Z][\-\{2\}(-gov)?-\[a-zA-Z]+-\d\{1\}:)?(\d\{12\}:)?(function:)?([a-zA-Z-A-Z\-9-\.]\+):((\$LATEST|[a-zA-Z\-9-\_]\+))?)

Required: Yes

**Qualifier**

Specify a version or alias to get the policy for that resource.


Pattern: (\[a-zA-Z\-9\$\-_]+)
Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

{   "Policy": "string",
   "RevisionId": "string"
}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**Policy**

The resource-based policy.

Type: String

**RevisionId**

A unique identifier for the current revision of the policy.

Type: String

Errors

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.
HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
GetProvisionedConcurrencyConfig

Retrieves the provisioned concurrency configuration for a function's alias or version.

Request Syntax

GET /2019-09-30/functions/FunctionName/provisioned-concurrency?Qualifier=Qualifier HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-\_]+)(((\$LATEST|[a-zA-Z0-9-\_]+))?)

Required: Yes

**Qualifier**

The version number or alias name.


Pattern: ([a-zA-Z0-9-\_]+)
Required: Yes

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

{
   "AllocatedProvisionedConcurrentExecutions": number,
   "AvailableProvisionedConcurrentExecutions": number,
   "LastModified": "string",
   "RequestedProvisionedConcurrentExecutions": number,
   "Status": "string",
   "StatusReason": "string"
}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

*AllocatedProvisionedConcurrentExecutions*

The amount of provisioned concurrency allocated. When a weighted alias is used during linear and canary deployments, this value fluctuates depending on the amount of concurrency that is provisioned for the function versions.

Type: Integer

Valid Range: Minimum value of 0.

*AvailableProvisionedConcurrentExecutions*

The amount of provisioned concurrency available.

Type: Integer

Valid Range: Minimum value of 0.
**LastModified**

The date and time that a user last updated the configuration, in ISO 8601 format.

Type: String

**RequestedProvisionedConcurrentExecutions**

The amount of provisioned concurrency requested.

Type: Integer

Valid Range: Minimum value of 1.

**Status**

The status of the allocation process.

Type: String

Valid Values: IN_PROGRESS | READY | FAILED

**StatusReason**

For failed allocations, the reason that provisioned concurrency could not be allocated.

Type: String

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ProvisionedConcurrencyConfigNotFoundException**

The specified configuration does not exist.

HTTP Status Code: 404

**ResourceNotFoundException**

The resource specified in the request does not exist.
HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
GetRuntimeManagementConfig

Retrieves the runtime management configuration for a function's version. If the runtime update mode is Manual, this includes the ARN of the runtime version and the runtime update mode. If the runtime update mode is Auto or Function update, this includes the runtime update mode and null is returned for the ARN. For more information, see Runtime updates.

Request Syntax

GET /2021-07-20/functions/FunctionName/runtime-management-config?Qualifier=Qualifier
HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

**Length Constraints:** Minimum length of 1. Maximum length of 170.

Pattern: (arn:aws[a-zA-Z-]*):lambda:)?([a-z]{2}(gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-._]+)(($LATEST|[a-zA-Z0-9-._]+))?

Required: Yes

**Qualifier**

Specify a version of the function. This can be $LATEST or a published version number. If no value is specified, the configuration for the $LATEST version is returned.

Pattern: ([a-zA-Z0-9$-_]+)

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

{
  "FunctionArn": "string",
  "RuntimeVersionArn": "string",
  "UpdateRuntimeOn": "string"
}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

FunctionArn

The Amazon Resource Name (ARN) of your function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:
  \d{12}:function:[a-zA-Z0-9-._]+(:($LATEST|[a-zA-Z0-9-._]+))?

RuntimeVersionArn

The ARN of the runtime the function is configured to use. If the runtime update mode is Manual, the ARN is returned, otherwise null is returned.

Type: String

Pattern: ^arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}((-gov)|(-iso(b?)))?-[a-z]+\d{1}::runtime:.+$

**UpdateRuntimeOn**

The current runtime update mode of the function.

Type: String

Valid Values: Auto | Manual | FunctionUpdate

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](https://docs.aws.amazon.com/lambda/latest/dg/limits.html).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](https://docs.aws.amazon.com/cli/index.html)
- [AWS SDK for .NET](https://docs.aws.amazon.com/sdk-for-net/v3/)

GetRuntimeManagementConfig
• AWS SDK for C++
• AWS SDK for Go
• AWS SDK for Java V2
• AWS SDK for JavaScript V3
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
Invoke

Invokes a Lambda function. You can invoke a function synchronously (and wait for the response), or asynchronously. By default, Lambda invokes your function synchronously (i.e. the InvocationType is RequestResponse). To invoke a function asynchronously, set InvocationType to Event. Lambda passes the ClientContext object to your function for synchronous invocations only.

For **synchronous invocation**, details about the function response, including errors, are included in the response body and headers. For either invocation type, you can find more information in the execution log and trace.

When an error occurs, your function may be invoked multiple times. Retry behavior varies by error type, client, event source, and invocation type. For example, if you invoke a function asynchronously and it returns an error, Lambda executes the function up to two more times. For more information, see Error handling and automatic retries in Lambda.

For **asynchronous invocation**, Lambda adds events to a queue before sending them to your function. If your function does not have enough capacity to keep up with the queue, events may be lost. Occasionally, your function may receive the same event multiple times, even if no error occurs. To retain events that were not processed, configure your function with a dead-letter queue.

The status code in the API response doesn't reflect function errors. Error codes are reserved for errors that prevent your function from executing, such as permissions errors, quota errors, or issues with your function's code and configuration. For example, Lambda returns TooManyRequestsException if running the function would cause you to exceed a concurrency limit at either the account level (ConcurrentInvocationLimitExceeded) or function level (ReservedFunctionConcurrentInvocationLimitExceeded).

For functions with a long timeout, your client might disconnect during synchronous invocation while it waits for a response. Configure your HTTP client, SDK, firewall, proxy, or operating system to allow for long connections with timeout or keep-alive settings.

This operation requires permission for the lambda:InvokeFunction action. For details on how to set up permissions for cross-account invocations, see Granting function access to other accounts.

**Request Syntax**

```plaintext
POST /2015-03-31/functions/FunctionName/invocations?Qualifier=Qualifier HTTP/1.1
```
X-Amz-Invocation-Type: *InvocationType*
X-Amz-Log-Type: *LogType*
X-Amz-Client-Context: *ClientContext*

**Payload**

**URI Request Parameters**

The request uses the following URI parameters.

**ClientContext**

Up to 3,583 bytes of base64-encoded data about the invoking client to pass to the function in the context object.

**FunctionName**

The name of the Lambda function, version, or alias.

**Name formats**

- **Function name** – my-function (name-only), my-function:v1 (with alias).
- **Partial ARN** – 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

**Length Constraints:** Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-._]+)(:(\$LATEST|[a-zA-Z0-9-._]+))?

**Required:** Yes

**InvocationType**

Choose from the following options.

- **RequestResponse** (default) – Invoke the function synchronously. Keep the connection open until the function returns a response or times out. The API response includes the function response and additional data.
• **Event** – Invoke the function asynchronously. Send events that fail multiple times to the function's dead-letter queue (if one is configured). The API response only includes a status code.

• **DryRun** – Validate parameter values and verify that the user or role has permission to invoke the function.

Valid Values: Event | RequestResponse | DryRun

**LogType**

Set to **Tail** to include the execution log in the response. Applies to synchronously invoked functions only.

Valid Values: None | Tail

**Qualifier**

Specify a version or alias to invoke a published version of the function.


Pattern: ([a-zA-Z0-9$_-]+)

**Request Body**

The request accepts the following binary data.

**Payload**

The JSON that you want to provide to your Lambda function as input.

You can enter the JSON directly. For example, `--payload '{ "key": "value" }'`. You can also specify a file path. For example, `--payload file://payload.json`.

**Response Syntax**

```
HTTP/1.1 StatusCode
X-Amz-Function-Error: FunctionError
X-Amz-Log-Result: LogResult
X-Amz-Executed-Version: ExecutedVersion
```

**Payload**
Response Elements

If the action is successful, the service sends back the following HTTP response.

**StatusCode**

The HTTP status code is in the 200 range for a successful request. For the RequestResponse invocation type, this status code is 200. For the Event invocation type, this status code is 202. For the DryRun invocation type, the status code is 204.

The response returns the following HTTP headers.

**ExecutedVersion**

The version of the function that executed. When you invoke a function with an alias, this indicates which version the alias resolved to.


Pattern: (\$LATEST|\d+)

**FunctionError**

If present, indicates that an error occurred during function execution. Details about the error are included in the response payload.

**LogResult**

The last 4 KB of the execution log, which is base64-encoded.

The response returns the following as the HTTP body.

**Payload**

The response from the function, or an error object.

**Errors**

**EC2AccessDeniedException**

Need additional permissions to configure VPC settings.
HTTP Status Code: 502

**EC2ThrottledException**

Amazon EC2 throttled AWS Lambda during Lambda function initialization using the execution role provided for the function.

HTTP Status Code: 502

**EC2UnexpectedException**

AWS Lambda received an unexpected Amazon EC2 client exception while setting up for the Lambda function.

HTTP Status Code: 502

**EFSIOException**

An error occurred when reading from or writing to a connected file system.

HTTP Status Code: 410

**EFSMountConnectivityException**

The Lambda function couldn't make a network connection to the configured file system.

HTTP Status Code: 408

**EFSMountFailureException**

The Lambda function couldn't mount the configured file system due to a permission or configuration issue.

HTTP Status Code: 403

**EFSMountTimeoutException**

The Lambda function made a network connection to the configured file system, but the mount operation timed out.

HTTP Status Code: 408

**ENILimitReachedException**

AWS Lambda couldn't create an elastic network interface in the VPC, specified as part of Lambda function configuration, because the limit for network interfaces has been reached. For more information, see [Lambda quotas](https://docs.aws.amazon.com/lambda/latest/dg/limits.html).
HTTP Status Code: 502

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**InvalidRequestContentException**

The request body could not be parsed as JSON.

HTTP Status Code: 400

**InvalidRuntimeException**

The runtime or runtime version specified is not supported.

HTTP Status Code: 502

**InvalidSecurityGroupIDException**

The security group ID provided in the Lambda function VPC configuration is not valid.

HTTP Status Code: 502

**InvalidSubnetIDException**

The subnet ID provided in the Lambda function VPC configuration is not valid.

HTTP Status Code: 502

**InvalidZipFileException**

AWS Lambda could not unzip the deployment package.

HTTP Status Code: 502

**KMSAccessDeniedException**

Lambda couldn't decrypt the environment variables because AWS KMS access was denied. Check the Lambda function's KMS permissions.

HTTP Status Code: 502

**KMSDisabledException**

Lambda couldn't decrypt the environment variables because the AWS KMS key used is disabled. Check the Lambda function's KMS key settings.
HTTP Status Code: 502

**KMSInvalidStateException**

Lambda couldn't decrypt the environment variables because the state of the AWS KMS key used is not valid for Decrypt. Check the function's KMS key settings.

HTTP Status Code: 502

**KMSNotFoundException**

Lambda couldn't decrypt the environment variables because the AWS KMS key was not found. Check the function's KMS key settings.

HTTP Status Code: 502

**RecursiveInvocationException**

Lambda has detected your function being invoked in a recursive loop with other AWS resources and stopped your function's invocation.

HTTP Status Code: 400

**RequestTooLargeException**

The request payload exceeded the Invoke request body JSON input quota. For more information, see Lambda quotas.

HTTP Status Code: 413

**ResourceConflictException**

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ResourceNotReadyException**

The function is inactive and its VPC connection is no longer available. Wait for the VPC connection to reestablish and try again.
HTTP Status Code: 502

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**SnapStartException**

The `afterRestore()` runtime hook encountered an error. For more information, check the Amazon CloudWatch logs.

HTTP Status Code: 400

**SnapStartNotReadyException**

Lambda is initializing your function. You can invoke the function when the function state becomes Active.

HTTP Status Code: 409

**SnapStartTimeoutException**

Lambda couldn't restore the snapshot within the timeout limit.

HTTP Status Code: 408

**SubnetIPAddressLimitReachedException**

AWS Lambda couldn't set up VPC access for the Lambda function because one or more configured subnets has no available IP addresses.

HTTP Status Code: 502

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

**UnsupportedMediaTypeException**

The content type of the Invoke request body is not JSON.

HTTP Status Code: 415
See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
InvokeAsync

This action has been deprecated.

⚠️ Important
For asynchronous function invocation, use Invoke.

Invokes a function asynchronously.

⚠️ Note
If you do use the InvokeAsync action, note that it doesn't support the use of X-Ray active tracing. Trace ID is not propagated to the function, even if X-Ray active tracing is turned on.

Request Syntax

POST /2014-11-13/functions/FunctionName/invoke-async/ HTTP/1.1
InvokeArgs

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?[a-z]{2}(gov)?-[a-z]+-[\d{1}]:?[\d{12}]:?(function:)?([a-zA-Z0-9-\._]+)(:$LATEST|[a-zA-Z0-9-\._]+)?

Required: Yes

Request Body

The request accepts the following binary data.

**InvokeArgs**

The JSON that you want to provide to your Lambda function as input.

Required: Yes

Response Syntax

HTTP/1.1 **Status**

Response Elements

If the action is successful, the service sends back the following HTTP response.

**Status**

The status code.

Errors

**InvalidRequestContentException**

The request body could not be parsed as JSON.

HTTP Status Code: 400

**InvalidRuntimeException**

The runtime or runtime version specified is not supported.
HTTP Status Code: 502

**ResourceConflictException**

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
InvokeWithResponseStream

Configure your Lambda functions to stream response payloads back to clients. For more information, see Configuring a Lambda function to stream responses.

This operation requires permission for the lambda:InvokeFunction action. For details on how to set up permissions for cross-account invocations, see Granting function access to other accounts.

Request Syntax

POST /2021-11-15/functions/\{FunctionName\}/response-streaming-invocations?Qualifier=\{Qualifier\} HTTP/1.1
X-Amz-Invocation-Type: InvocationType
X-Amz-Log-Type: LogType
X-Amz-Client-Context: ClientContext

Payload

URI Request Parameters

The request uses the following URI parameters.

ClientContext

Up to 3,583 bytes of base64-encoded data about the invoking client to pass to the function in the context object.

FunctionName

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

**Pattern**: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_.]+)(:(\$LATEST|[a-zA-Z0-9-_.]+))?  

Required: Yes

**InvocationType**

Use one of the following options:

- **RequestResponse** (default) – Invoke the function synchronously. Keep the connection open until the function returns a response or times out. The API operation response includes the function response and additional data.
- **DryRun** – Validate parameter values and verify that the IAM user or role has permission to invoke the function.

Valid Values: RequestResponse | DryRun

**LogType**

Set to Tail to include the execution log in the response. Applies to synchronously invoked functions only.

Valid Values: None | Tail

**Qualifier**

The alias name.


**Pattern**: ( | [a-zA-Z0-9-9$-_.]+)

**Request Body**

The request accepts the following binary data.

**Payload**

The JSON that you want to provide to your Lambda function as input.

You can enter the JSON directly. For example, --payload '{ "key": "value" }'. You can also specify a file path. For example, --payload file://payload.json.
Response Syntax

HTTP/1.1 **StatusCode**
X-Amz-Executed-Version: **ExecutedVersion**
Content-Type: **ResponseStreamContentType**

```json
{
    "InvokeComplete": {
        "ErrorCode": "string",
        "ErrorDetails": "string",
        "LogResult": "string"
    },
    "PayloadChunk": {
        "Payload": "blob"
    }
}
```

Response Elements

If the action is successful, the service sends back the following HTTP response.

**StatusCode**

For a successful request, the HTTP status code is in the 200 range. For the RequestResponse invocation type, this status code is 200. For the DryRun invocation type, this status code is 204.

The response returns the following HTTP headers.

**ExecutedVersion**

The version of the function that executed. When you invoke a function with an alias, this indicates which version the alias resolved to.


Pattern: (\$LATEST|[0-9]+)

**ResponseStreamContentType**

The type of data the stream is returning.
The following data is returned in JSON format by the service.

**InvokeComplete**

An object that's returned when the stream has ended and all the payload chunks have been returned.

Type: `InvokeWithResponseStreamCompleteEvent` object

**PayloadChunk**

A chunk of the streamed response payload.

Type: `InvokeResponseStreamUpdate` object

**Errors**

**EC2AccessDeniedException**

Need additional permissions to configure VPC settings.

HTTP Status Code: 502

**EC2ThrottledException**

Amazon EC2 throttled AWS Lambda during Lambda function initialization using the execution role provided for the function.

HTTP Status Code: 502

**EC2UnexpectedException**

AWS Lambda received an unexpected Amazon EC2 client exception while setting up for the Lambda function.

HTTP Status Code: 502

**EFSIOException**

An error occurred when reading from or writing to a connected file system.

HTTP Status Code: 410

**EFSMountConnectivityException**

The Lambda function couldn't make a network connection to the configured file system.
HTTP Status Code: 408

EFSMountFailureException

The Lambda function couldn't mount the configured file system due to a permission or configuration issue.

HTTP Status Code: 403

EFSMountTimeoutException

The Lambda function made a network connection to the configured file system, but the mount operation timed out.

HTTP Status Code: 408

ENILimitReachedException

AWS Lambda couldn't create an elastic network interface in the VPC, specified as part of Lambda function configuration, because the limit for network interfaces has been reached. For more information, see Lambda quotas.

HTTP Status Code: 502

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

InvalidRequestContentException

The request body could not be parsed as JSON.

HTTP Status Code: 400

InvalidRuntimeException

The runtime or runtime version specified is not supported.

HTTP Status Code: 502

InvalidSecurityGroupIDException

The security group ID provided in the Lambda function VPC configuration is not valid.

HTTP Status Code: 502
InvalidSubnetIDException

The subnet ID provided in the Lambda function VPC configuration is not valid.

HTTP Status Code: 502

InvalidZipFileException

AWS Lambda could not unzip the deployment package.

HTTP Status Code: 502

KMSAccessDeniedException

Lambda couldn't decrypt the environment variables because AWS KMS access was denied. Check the Lambda function's KMS permissions.

HTTP Status Code: 502

KMSDisabledException

Lambda couldn't decrypt the environment variables because the AWS KMS key used is disabled. Check the Lambda function's KMS key settings.

HTTP Status Code: 502

KMSInvalidStateException

Lambda couldn't decrypt the environment variables because the state of the AWS KMS key used is not valid for Decrypt. Check the function's KMS key settings.

HTTP Status Code: 502

KMSNotFoundException

Lambda couldn't decrypt the environment variables because the AWS KMS key was not found. Check the function's KMS key settings.

HTTP Status Code: 502

RecursiveInvocationException

Lambda has detected your function being invoked in a recursive loop with other AWS resources and stopped your function's invocation.

HTTP Status Code: 400
RequestTooLargeException

The request payload exceeded the Invoke request body JSON input quota. For more information, see Lambda quotas.

HTTP Status Code: 413

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ResourceNotReadyException

The function is inactive and its VPC connection is no longer available. Wait for the VPC connection to reestablish and try again.

HTTP Status Code: 502

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

SnapStartException

The afterRestore() runtime hook encountered an error. For more information, check the Amazon CloudWatch logs.

HTTP Status Code: 400

SnapStartNotReadyException

Lambda is initializing your function. You can invoke the function when the function state becomes Active.

HTTP Status Code: 409
SnapStartTimeoutException

Lambda couldn't restore the snapshot within the timeout limit.

HTTP Status Code: 408

SubnetIPAddressLimitReachedException

AWS Lambda couldn't set up VPC access for the Lambda function because one or more configured subnets has no available IP addresses.

HTTP Status Code: 502

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

UnsupportedMediaTypeException

The content type of the Invoke request body is not JSON.

HTTP Status Code: 415

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
**ListAliases**

Returns a list of aliases for a Lambda function.

**Request Syntax**

```
GET /2015-03-31/functions/FunctionName/aliases?
FunctionVersion=FunctionVersion&Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

**URI Request Parameters**

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** - MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z][2]{-gov}?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-]+)(:\($LATEST|[a-zA-Z0-9-_]\))?(

Required: Yes

**FunctionVersion**

Specify a function version to only list aliases that invoke that version.


Pattern: (\$LATEST|[0-9]+)
### Marker

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

### MaxItems

Limit the number of aliases returned.

Valid Range: Minimum value of 1. Maximum value of 10000.

### Request Body

The request does not have a request body.

### Response Syntax

```
HTTP/1.1 200
Content-type: application/json
{
    "Aliases": [
        {
            "AliasArn": "string",
            "Description": "string",
            "FunctionVersion": "string",
            "Name": "string",
            "RevisionId": "string",
            "RoutingConfig": {
                "AdditionalVersionWeights": {
                    "string": number
                }
            }
        },
        ...
    ],
    "NextMarker": "string"
}
```

### Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.
**Aliases**

A list of aliases.

Type: Array of `AliasConfiguration` objects

**NextMarker**

The pagination token that's included if more results are available.

Type: String

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](https://docs.aws.amazon.com/lambda/latest/dg/security-quotas.html).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](https://docs.aws.amazon.com/cli/latest/reference/lambda/list-aliases.html)
• AWS SDK for .NET
• AWS SDK for C++
• AWS SDK for Go
• AWS SDK for Java V2
• AWS SDK for JavaScript V3
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
ListCodeSigningConfigs

Returns a list of code signing configurations. A request returns up to 10,000 configurations per call. You can use the MaxItems parameter to return fewer configurations per call.

Request Syntax

```
GET /2020-04-22/code-signing-configs/?Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

**Marker**

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

**MaxItems**

Maximum number of items to return.

Valid Range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "CodeSigningConfigs": [
        {
            "AllowedPublishers": {
                "SigningProfileVersionArns": [ "string" ]
            },
            "CodeSigningConfigArn": "string",
            ...
        }
    ]
}
```
"CodeSigningConfigId": "string",
"CodeSigningPolicies": {
    "UntrustedArtifactOnDeployment": "string"
},
"Description": "string",
"LastModified": "string"
}]
"NextMarker": "string"
}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**CodeSigningConfigs**

The code signing configurations

Type: Array of CodeSigningConfig objects

**NextMarker**

The pagination token that's included if more results are available.

Type: String

Errors

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500
See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
ListEventSourceMappings

Lists event source mappings. Specify an EventSourceArn to show only event source mappings for a single event source.

Request Syntax

GET /2015-03-31/event-source-mappings/?
EventSourceArn=EventSourceArn&FunctionName=FunctionName&Marker=Marker&MaxItems=MaxItems
HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**EventSourceArn**

The Amazon Resource Name (ARN) of the event source.

- **Amazon Kinesis** – The ARN of the data stream or a stream consumer.
- **Amazon DynamoDB Streams** – The ARN of the stream.
- **Amazon Simple Queue Service** – The ARN of the queue.
- **Amazon Managed Streaming for Apache Kafka** – The ARN of the cluster or the ARN of the VPC connection (for cross-account event source mappings).
- **Amazon MQ** – The ARN of the broker.
- **Amazon DocumentDB** – The ARN of the DocumentDB change stream.

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-]+\d{12}):\d{12}?:(.*)

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** – MyFunction.
- **Partial ARN** – 123456789012:function:MyFunction.
The length constraint applies only to the full ARN. If you specify only the function name, it's limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-\_]+)(:(\$LATEST|[a-zA-Z0-9-\_]+))+)

**Marker**

A pagination token returned by a previous call.

**MaxItems**

The maximum number of event source mappings to return. Note that ListEventSourceMappings returns a maximum of 100 items in each response, even if you set the number higher.

Valid Range: Minimum value of 1. Maximum value of 10000.

**Request Body**

The request does not have a request body.

**Response Syntax**

HTTP/1.1 200
Content-type: application/json

```json
{
   "EventSourceMappings": [
      {
         "AmazonManagedKafkaEventSourceConfig": {
            "ConsumerGroupId": "string"
         },
         "BatchSize": number,
         "BisectBatchOnFunctionError": boolean,
         "DestinationConfig": {
            "OnFailure": {
               "Destination": "string"
            },
            "OnSuccess": {
               "Destination": "string"
            }
         }
      }
   ]
}
```

ListEventSourceMappings
"DocumentDBEventSourceConfig": {
    "CollectionName": "string",
    "DatabaseName": "string",
    "FullDocument": "string"
},
"EventSourceArn": "string",
"FilterCriteria": {
    "Filters": [
        {
            "Pattern": "string"
        }
    ]
},
"FunctionArn": "string",
"FunctionResponseTypes": [ "string" ],
"LastModified": number,
"LastProcessingResult": "string",
"MaximumBatchingWindowInSeconds": number,
"MaximumRecordAgeInSeconds": number,
"MaximumRetryAttempts": number,
"ParallelizationFactor": number,
"Queues": [ "string" ],
"ScalingConfig": {
    "MaximumConcurrency": number
},
"SelfManagedEventSource": {
    "Endpoints": {
        "string": [ "string" ]
    }
},
"SelfManagedKafkaEventSourceConfig": {
    "ConsumerGroupId": "string"
},
"SourceAccessConfigurations": [
    {
        "Type": "string",
        "URI": "string"
    }
],
"StartingPosition": "string",
"StartingPositionTimestamp": number,
"State": "string",
"StateTransitionReason": "string",
"Topics": [ "string" ]
Response Elements

If the action is successful, the service sends back an HTTP 200 response. The following data is returned in JSON format by the service.

**EventSourceMappings**

A list of event source mappings.

Type: Array of `EventSourceMappingConfiguration` objects

**NextMarker**

A pagination token that's returned when the response doesn't contain all event source mappings.

Type: String

Errors

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500
TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
ListFunctionEventInvokeConfigs

Retrieves a list of configurations for asynchronous invocation for a function.

To configure options for asynchronous invocation, use PutFunctionEventInvokeConfig.

Request Syntax

GET /2019-09-25/functions/FunctionName/event-invoke-config/list?Marker=Marker&MaxItems=MaxItems HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

FunctionName

The name of the Lambda function.

Name formats

- **Function name** - my-function.
- **Partial ARN** - 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-\_]*)?(:($LATEST|[a-zA-Z0-9-\_]+))?  

Required: Yes

Marker

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

MaxItems

The maximum number of configurations to return.

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

{
   "FunctionEventInvokeConfigs": [
      {
         "DestinationConfig": {
            "OnFailure": {
               "Destination": "string"
            },
            "OnSuccess": {
               "Destination": "string"
            }
         },
         "FunctionArn": "string",
         "LastModified": number,
         "MaximumEventAgeInSeconds": number,
         "MaximumRetryAttempts": number
      }
   ],
   "NextMarker": "string"
}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**FunctionEventInvokeConfigs**

A list of configurations.

Type: Array of [FunctionEventInvokeConfig](#) objects
**NextMarker**

The pagination token that's included if more results are available.

Type: String

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
**ListFunctions**

Returns a list of Lambda functions, with the version-specific configuration of each. Lambda returns up to 50 functions per call.

Set FunctionVersion to ALL to include all published versions of each function in addition to the unpublished version.

⚠️ **Note**

The ListFunctions operation returns a subset of the FunctionConfiguration fields. To get the additional fields (State, StateReasonCode, StateReason, LastUpdateStatus, LastUpdateStatusReason, LastUpdateStatusReasonCode, RuntimeVersionConfig) for a function or version, use GetFunction.

**Request Syntax**

```
GET /2015-03-31/functions/?
FunctionVersion=FunctionVersion&Marker=Marker&MasterRegion=MasterRegion&MaxItems=MaxItems
HTTP/1.1
```

**URI Request Parameters**

The request uses the following URI parameters.

**FunctionVersion**

Set to ALL to include entries for all published versions of each function.

Valid Values: ALL

**Marker**

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

**MasterRegion**

For Lambda@Edge functions, the AWS Region of the master function. For example, us-east-1 filters the list of functions to include only Lambda@Edge functions replicated from a master function in US East (N. Virginia). If specified, you must set FunctionVersion to ALL.
Pattern: ALL\[[a-z]\{2\}(\-gov)?-\[a-z]+\-\d\]{1}\n
MaxItems

The maximum number of functions to return in the response. Note that ListFunctions returns a maximum of 50 items in each response, even if you set the number higher.

Valid Range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

```json
{
   "Functions": [
      {
         "Architectures": [ "string" ],
         "CodeSha256": "string",
         "CodeSize": number,
         "DeadLetterConfig": {
            "TargetArn": "string"
         },
         "Description": "string",
         "Environment": {
            "Error": {
               "ErrorCode": "string",
               "Message": "string"
            },
            "Variables": {
               "string": "string"
            }
         },
         "EphemeralStorage": {
            "Size": number
         },
         "FileSystemConfigs": [
            {
               "Arn": "string"
            }
         ]
      }
   ]
}
```
"LocalMountPath": "string",
]
"FunctionArn": "string",
"FunctionName": "string",
"Handler": "string",
"ImageConfigResponse": {
  "Error": {
    "ErrorCode": "string",
    "Message": "string"
  },
  "ImageConfig": {
    "Command": [ "string" ],
    "EntryPoint": [ "string" ],
    "WorkingDirectory": "string"
  }
},
"KMSKeyArn": "string",
"LastModified": "string",
"LastUpdateStatus": "string",
"LastUpdateStatusReason": "string",
"LastUpdateStatusReasonCode": "string",
"Layers": [
  {
    "Arn": "string",
    "CodeSize": number,
    "SigningJobArn": "string",
    "SigningProfileVersionArn": "string"
  }
],
"LoggingConfig": {
  "ApplicationLogLevel": "string",
  "LogFormat": "string",
  "LogGroup": "string",
  "SystemLogLevel": "string"
},
"MasterArn": "string",
"MemorySize": number,
"PackageType": "string",
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"RuntimeVersionConfig": {
  "Error": {

"ErrorCode": "string",
"Message": "string"
},
"RuntimeVersionArn": "string"
},
"SigningJobArn": "string",
"SigningProfileVersionArn": "string",
"SnapStart": {
   "ApplyOn": "string",
   "OptimizationStatus": "string"
},
"State": "string",
"StateReason": "string",
"StateReasonCode": "string",
"Timeout": number,
"TracingConfig": {
   "Mode": "string"
},
"Version": "string",
"VpcConfig": {
   "Ipv6AllowedForDualStack": boolean,
   "SecurityGroupIds": [ "string" ],
   "SubnetIds": [ "string" ],
   "VpcId": "string"
}
],
"NextMarker": "string"}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Functions

A list of Lambda functions.

Type: Array of FunctionConfiguration objects

NextMarker

The pagination token that's included if more results are available.
Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
ListFunctionsByCodeSigningConfig

List the functions that use the specified code signing configuration. You can use this method prior to deleting a code signing configuration, to verify that no functions are using it.

Request Syntax

GET /2020-04-22/code-signing-configs/CodeSigningConfigArn/functions?
Marker=Marker&MaxItems=MaxItems HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**CodeSigningConfigArn**

The Amazon Resource Name (ARN) of the code signing configuration.

Length Constraints: Maximum length of 200.

Pattern:

```
arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}((-gov)|(-iso(b?)))?-[a-zA-Z]+-\d{1}:\d{12}:code-signing-config:csc-[a-z0-9]{17}
```

Required: Yes

**Marker**

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

**MaxItems**

Maximum number of items to return.

Valid Range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

{
  "FunctionArns": [ "string" ],
  "NextMarker": "string"
}

**Response Elements**

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**FunctionArns**

The function ARNs.

Type: Array of strings

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}: \d{12}:function:[a-zA-Z0-9-9-\_]+((\$LATEST|[a-zA-Z0-9-9-\_]+))?`

**NextMarker**

The pagination token that's included if more results are available.

Type: String

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.
HTTP Status Code: 500

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
ListFunctionUrlConfigs

Returns a list of Lambda function URLs for the specified function.

Request Syntax

```
GET /2021-10-31/functions/FunctionName/urls?Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?(\[a-zA-Z]{2}(-gov)?-[a-zA-Z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-\_]+)(:(\$LATEST|[a-zA-Z0-9-\_]\+))?

Required: Yes

**Marker**

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

**MaxItems**

The maximum number of function URLs to return in the response. Note that ListFunctionUrlConfigs returns a maximum of 50 items in each response, even if you set the number higher.

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

```json
{
    "FunctionUrlConfigs": [
        {
            "AuthType": "string",
            "Cors": {
                "AllowCredentials": boolean,
                "AllowHeaders": [ "string" ],
                "AllowMethods": [ "string" ],
                "AllowOrigins": [ "string" ],
                "ExposeHeaders": [ "string" ],
                "MaxAge": number
            },
            "CreationTime": "string",
            "FunctionArn": "string",
            "FunctionUrl": "string",
            "InvokeMode": "string",
            "LastModifiedTime": "string"
        }
    ],
    "NextMarker": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**FunctionUrlConfigs**

A list of function URL configurations.
Type: Array of `FunctionUrlConfig` objects

**NextMarker**

The pagination token that's included if more results are available.

Type: String

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](https://docs.aws.amazon.com/lambda/latest/dg/quotas.html).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](https://aws.amazon.com/cli/)
- [AWS SDK for .NET](https://aws.amazon.com/sdk-for-net/)
- [AWS SDK for C++](https://aws.amazon.com/sdk-for-cpp/)
- [AWS SDK for Go](https://aws.amazon.com/sdk-for-go/)

• AWS SDK for Java V2
• AWS SDK for JavaScript V3
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
ListLayers

Lists AWS Lambda layers and shows information about the latest version of each. Specify a runtime identifier to list only layers that indicate that they're compatible with that runtime. Specify a compatible architecture to include only layers that are compatible with that instruction set architecture.

Request Syntax

GET /2018-10-31/layers?
CompatibleArchitecture=CompatibleArchitecture&CompatibleRuntime=CompatibleRuntime&Marker=Marker
HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**CompatibleArchitecture**

The compatible instruction set architecture.

Valid Values: x86_64 | arm64

**CompatibleRuntime**

A runtime identifier. For example, go1.x.

The following list includes deprecated runtimes. For more information, see Runtime deprecation policy.

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17 | ruby3.2 | python3.11 | nodejs20.x | provided.al2023 | python3.12 | java21

**Marker**

A pagination token returned by a previous call.
MaxItems

The maximum number of layers to return.


Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

{
  "Layers": [
    {
      "LatestMatchingVersion": {
        "CompatibleArchitectures": [ "string" ],
        "CompatibleRuntimes": [ "string" ],
        "CreatedDate": "string",
        "Description": "string",
        "LayerVersionArn": "string",
        "LicenseInfo": "string",
        "Version": number
      },
      "LayerArn": "string",
      "LayerName": "string"
    }
  ],
  "NextMarker": "string"
}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Layers

A list of function layers.
Type: Array of `LayersListItem` objects

**NextMarker**

A pagination token returned when the response doesn't contain all layers.

Type: String

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](https://docs.aws.amazon.com/lambda/latest/dg/limits.html).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](https://aws.amazon.com/cli/)
- [AWS SDK for .NET](https://aws.amazon.com/sdk-for-net/)
- [AWS SDK for C++](https://aws.amazon.com/sdk-for-cpp/)
- [AWS SDK for Go](https://aws.amazon.com/sdk-for-go/)
- [AWS SDK for Java V2](https://aws.amazon.com/sdk-for-java/)
- [AWS SDK for JavaScript V3](https://aws.amazon.com/sdk-for-js/)
- [AWS SDK for PHP V3](https://aws.amazon.com/sdk-for-php/)
- [AWS SDK for Python](https://aws.amazon.com/sdk-for-python/)
• **AWS SDK for Ruby V3**
ListLayerVersions

Lists the versions of an AWS Lambda layer. Versions that have been deleted aren't listed. Specify a runtime identifier to list only versions that indicate that they're compatible with that runtime. Specify a compatible architecture to include only layer versions that are compatible with that architecture.

Request Syntax

GET /2018-10-31/layers/LayerName/versions?
CompatibleArchitecture=CompatibleArchitecture&CompatibleRuntime=CompatibleRuntime&Marker=Marker
HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

CompatibleArchitecture

The compatible instruction set architecture.

Valid Values: x86_64 | arm64

CompatibleRuntime

A runtime identifier. For example, go1.x.

The following list includes deprecated runtimes. For more information, see Runtime deprecation policy.

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17 | ruby3.2 | python3.11 | nodejs20.x | provided.al2023 | python3.12 | java21

LayerName

The name or Amazon Resource Name (ARN) of the layer.
Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-]+)+|[a-zA-Z0-9-]+)

Required: Yes

**Marker**

A pagination token returned by a previous call.

**MaxItems**

The maximum number of versions to return.


**Request Body**

The request does not have a request body.

**Response Syntax**

HTTP/1.1 200
Content-type: application/json

```
{
    "LayerVersions": [
    {
        "CompatibleArchitectures": [ "string" ],
        "CompatibleRuntimes": [ "string" ],
        "CreatedDate": "string",
        "Description": "string",
        "LayerVersionArn": "string",
        "LicenseInfo": "string",
        "Version": number
    }
    ],
    "NextMarker": "string"
}
```

**Response Elements**

If the action is successful, the service sends back an HTTP 200 response.
The following data is returned in JSON format by the service.

**LayerVersions**

A list of versions.

Type: Array of [LayerVersionsListItem](#) objects

**NextMarker**

A pagination token returned when the response doesn't contain all versions.

Type: String

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:
• **AWS Command Line Interface**
• **AWS SDK for .NET**
• **AWS SDK for C++**
• **AWS SDK for Go**
• **AWS SDK for Java V2**
• **AWS SDK for JavaScript V3**
• **AWS SDK for PHP V3**
• **AWS SDK for Python**
• **AWS SDK for Ruby V3**
ListProvisionedConcurrencyConfigs

Retrieves a list of provisioned concurrency configurations for a function.

Request Syntax

```
GET /2019-09-30/functions/FunctionName/provisioned-concurrency?
List=ALL&Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:\$LATEST|[a-zA-Z0-9-_:]+)?

Required: Yes

**Marker**

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

**MaxItems**

Specify a number to limit the number of configurations returned.

**Request Body**

The request does not have a request body.

**Response Syntax**

```json
HTTP/1.1 200
Content-type: application/json

{
    "NextMarker": "string",
    "ProvisionedConcurrencyConfigs": [
        {
            "AllocatedProvisionedConcurrentExecutions": number,
            "AvailableProvisionedConcurrentExecutions": number,
            "FunctionArn": "string",
            "LastModified": "string",
            "RequestedProvisionedConcurrentExecutions": number,
            "Status": "string",
            "StatusReason": "string"
        }
    ]
}
```

**Response Elements**

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**NextMarker**

The pagination token that's included if more results are available.

Type: String

**ProvisionedConcurrencyConfigs**

A list of provisioned concurrency configurations.

Type: Array of ProvisionedConcurrencyConfigListItem objects
Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
ListTags

Returns a function's tags. You can also view tags with GetFunction.

Request Syntax

GET /2017-03-31/tags/ARN HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

**ARN**

The function's Amazon Resource Name (ARN). Note: Lambda does not support adding tags to aliases or versions.

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:
\d{12}:function:[a-zA-Z0-9-9-\_]+(:($LATEST|[a-zA-Z0-9-9-\_]+))+)

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 200
Content-type: application/json

```
{
   "Tags": {
      "string" : "string"
   }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.
The following data is returned in JSON format by the service.

**Tags**

The function's tags.

Type: String to string map

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
• AWS SDK for Go
• AWS SDK for Java V2
• AWS SDK for JavaScript V3
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
ListVersionsByFunction

Returns a list of versions, with the version-specific configuration of each. Lambda returns up to 50 versions per call.

Request Syntax

```
GET /2015-03-31/functions/FunctionName/versions?Marker=Marker&MaxItems=MaxItems
HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** - MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.


Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-._]+)(:\($LATEST|[a-zA-Z0-9-._]+$)?)?

Required: Yes

**Marker**

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

**MaxItems**

The maximum number of versions to return. Note that ListVersionsByFunction returns a maximum of 50 items in each response, even if you set the number higher.
Valid Range: Minimum value of 1. Maximum value of 10000.

**Request Body**

The request does not have a request body.

**Response Syntax**

```
HTTP/1.1 200
Content-type: application/json

{
    "NextMarker": "string",
    "Versions": [
        {
            "Architectures": [ "string" ],
            "CodeSha256": "string",
            "CodeSize": number,
            "DeadLetterConfig": {
                "TargetArn": "string"
            },
            "Description": "string",
            "Environment": {
                "Error": {
                    "ErrorCode": "string",
                    "Message": "string"
                },
                "Variables": {
                    "string": "string"
                }
            },
            "EphemeralStorage": {
                "Size": number
            },
            "FileSystemConfigs": [
                {
                    "Arn": "string",
                    "LocalMountPath": "string"
                }
            ],
            "FunctionArn": "string",
            "FunctionName": "string",
            "Handler": "string",
```
"ImageConfigResponse": {
    "Error": {
        "ErrorCode": "string",
        "Message": "string"
    },
    "ImageConfig": {
        "Command": [ "string" ],
        "EntryPoint": [ "string" ],
        "WorkingDirectory": "string"
    }
},
"KMSKeyArn": "string",
"LastModified": "string",
"LastUpdateStatus": "string",
"LastUpdateStatusReason": "string",
"LastUpdateStatusReasonCode": "string",
"Layers": [
    {
        "Arn": "string",
        "CodeSize": number,
        "SigningJobArn": "string",
        "SigningProfileVersionArn": "string"
    }
],
"LoggingConfig": {
    "ApplicationLogLevel": "string",
    "LogFormat": "string",
    "LogGroup": "string",
    "SystemLogLevel": "string"
},
"MasterArn": "string",
"MemorySize": number,
"PackageType": "string",
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"RuntimeVersionConfig": {
    "Error": {
        "ErrorCode": "string",
        "Message": "string"
    },
    "RuntimeVersionArn": "string"
},
"SigningJobArn": "string"
"SigningProfileArn": "string",
"SnapStart": {
    "ApplyOn": "string",
    "OptimizationStatus": "string"
},
"State": "string",
"StateReason": "string",
"StateReasonCode": "string",
"Timeout": number,
"TracingConfig": {
    "Mode": "string"
},
"Version": "string",
"VpcConfig": {
    "Ipv6AllowedForDualStack": boolean,
    "SecurityGroupIds": [ "string" ],
    "SubnetIds": [ "string" ],
    "VpcId": "string"
}
}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

NextMarker

The pagination token that's included if more results are available.

Type: String

Versions

A list of Lambda function versions.

Type: Array of FunctionConfiguration objects
Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
PublishLayerVersion

Creates an AWS Lambda layer from a ZIP archive. Each time you call PublishLayerVersion with the same layer name, a new version is created.

Add layers to your function with CreateFunction or UpdateFunctionConfiguration.

Request Syntax

POST /2018-10-31/layers/{LayerName}/versions HTTP/1.1
Content-type: application/json

{
  "CompatibleArchitectures": [ "string" ],
  "CompatibleRuntimes": [ "string" ],
  "Content": {
    "S3Bucket": "string",
    "S3Key": "string",
    "S3ObjectVersion": "string",
    "ZipFile": "blob"
  },
  "Description": "string",
  "LicenseInfo": "string"
}

URI Request Parameters

The request uses the following URI parameters.

LayerName

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-\_]\+)\([a-zA-Z0-9-\_]\)
Request Body

The request accepts the following data in JSON format.

**CompatibleArchitectures**

A list of compatible instruction set architectures.

Type: Array of strings

Array Members: Maximum number of 2 items.

Valid Values: x86_64 | arm64

Required: No

**CompatibleRuntimes**

A list of compatible function runtimes. Used for filtering with ListLayers and ListLayerVersions.

The following list includes deprecated runtimes. For more information, see Runtime deprecation policy.

Type: Array of strings

Array Members: Maximum number of 15 items.

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17 | ruby3.2 | python3.11 | nodejs20.x | provided.al2023 | python3.12 | java21

Required: No

**Content**

The function layer archive.

Type: LayerVersionContentInput object

Required: Yes
Description

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

LicenseInfo

The layer's software license. It can be any of the following:

- An SPDX license identifier. For example, MIT.
- The URL of a license hosted on the internet. For example, https://opensource.org/licenses/MIT.
- The full text of the license.

Type: String

Length Constraints: Maximum length of 512.

Required: No

Response Syntax

HTTP/1.1 201
Content-type: application/json

```json
{
   "CompatibleArchitectures": [ "string" ],
   "CompatibleRuntimes": [ "string" ],
   "Content": {
      "CodeSha256": "string",
      "CodeSize": number,
      "Location": "string",
      "SigningJobArn": "string",
      "SigningProfileVersionArn": "string"
   },
   "CreatedDate": "string",
   "Description": "string",
   "LayerArn": "string"
}
```
"LayerVersionArn": "string",
"LicenseInfo": "string",
"Version": number
}

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

**CompatibleArchitectures**

A list of compatible instruction set architectures.

Type: Array of strings

Array Members: Maximum number of 2 items.

Valid Values: x86_64 | arm64

**CompatibleRuntimes**

The layer's compatible runtimes.

The following list includes deprecated runtimes. For more information, see Runtime deprecation policy.

Type: Array of strings

Array Members: Maximum number of 15 items.

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17 | ruby3.2 | python3.11 | nodejs20.x | provided.al2023 | python3.12 | java21

**Content**

Details about the layer version.
Type: LayerVersionContentOutput object

**CreatedDate**

The date that the layer version was created, in ISO-8601 format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

**Description**

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

**LayerArn**

The ARN of the layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_.]+:

**LayerVersionArn**

The ARN of the layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_.]+:[0-9]+:

**LicenseInfo**

The layer's software license.

Type: String

Length Constraints: Maximum length of 512.
Version

The version number.

Type: Long

Errors

CodeStorageExceededException

Your AWS account has exceeded its maximum total code size. For more information, see Lambda quotas.

HTTP Status Code: 400

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:
• **AWS Command Line Interface**
• **AWS SDK for .NET**
• **AWS SDK for C++**
• **AWS SDK for Go**
• **AWS SDK for Java V2**
• **AWS SDK for JavaScript V3**
• **AWS SDK for PHP V3**
• **AWS SDK for Python**
• **AWS SDK for Ruby V3**
PublishVersion

Creates a version from the current code and configuration of a function. Use versions to create a snapshot of your function code and configuration that doesn't change.

AWS Lambda doesn't publish a version if the function's configuration and code haven't changed since the last version. Use UpdateFunctionCode or UpdateFunctionConfiguration to update the function before publishing a version.

Clients can invoke versions directly or with an alias. To create an alias, use CreateAlias.

Request Syntax

POST /2015-03-31/functions/FunctionName/versions HTTP/1.1
Content-type: application/json

{
   "CodeSha256": "string",
   "Description": "string",
   "RevisionId": "string"
}

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** - MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

**Length Constraints:** Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-zA-Z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_.]+)(:\($LATEST|[a-zA-Z0-9-_.]+))?
Required: Yes

**Request Body**

The request accepts the following data in JSON format.

**CodeSha256**

Only publish a version if the hash value matches the value that's specified. Use this option to avoid publishing a version if the function code has changed since you last updated it. You can get the hash for the version that you uploaded from the output of `UpdateFunctionCode`.

Type: String

Required: No

**Description**

A description for the version to override the description in the function configuration.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

**RevisionId**

Only update the function if the revision ID matches the ID that's specified. Use this option to avoid publishing a version if the function configuration has changed since you last updated it.

Type: String

Required: No

**Response Syntax**

```
HTTP/1.1 201
Content-type: application/json

{
   "Architectures": [ "string" ],
   "CodeSha256": "string",
   "CodeSize": number,
}
```
"DeadLetterConfig": {
  "TargetArn": "string"
},
"Description": "string",
"Environment": {
  "Error": {
    "ErrorCode": "string",
    "Message": "string"
  },
  "Variables": {
    "string": "string"
  }
},
"EphemeralStorage": {
  "Size": number
},
"FileSystemConfigs": [
  {
    "Arn": "string",
    "LocalMountPath": "string"
  }
],
"FunctionArn": "string",
"FunctionName": "string",
"Handler": "string",
"ImageConfigResponse": {
  "Error": {
    "ErrorCode": "string",
    "Message": "string"
  },
  "ImageConfig": {
    "Command": [ "string" ],
    "EntryPoint": [ "string" ],
    "WorkingDirectory": "string"
  }
},
"KMSKeyArn": "string",
"LastModified": "string",
"LastUpdateStatus": "string",
"LastUpdateStatusReason": "string",
"LastUpdateStatusReasonCode": "string",
"Layers": [
  {
    "Arn": "string"
  }
]
"CodeSize": number,
"SigningJobArn": "string",
"SigningProfileVersionArn": "string"
}
],
"LoggingConfig": {
  "ApplicationLogLevel": "string",
  "LogFormat": "string",
  "LogGroup": "string",
  "SystemLogLevel": "string"
},
"MasterArn": "string",
"MemorySize": number,
"PackageType": "string",
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"RuntimeVersionConfig": {
  "Error": {
    "ErrorCode": "string",
    "Message": "string"
  },
  "RuntimeVersionArn": "string"
},
"SigningJobArn": "string",
"SigningProfileVersionArn": "string",
"SnapStart": {
  "ApplyOn": "string",
  "OptimizationStatus": "string"
},
"State": "string",
"StateReason": "string",
"StateReasonCode": "string",
"Timeout": number,
"TracingConfig": {
  "Mode": "string"
},
"Version": "string",
"VpcConfig": {
  "Ipv6AllowedForDualStack": boolean,
  "SecurityGroupIds": [ "string" ],
  "SubnetIds": [ "string" ],
  "VpcId": "string"
}
Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

**Architectures**

The instruction set architecture that the function supports. Architecture is a string array with one of the valid values. The default architecture value is x86_64.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: x86_64 | arm64

**CodeSha256**

The SHA256 hash of the function's deployment package.

Type: String

**CodeSize**

The size of the function's deployment package, in bytes.

Type: Long

**DeadLetterConfig**

The function's dead letter queue.

Type: DeadLetterConfig object

**Description**

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.
**Environment**

The function's environment variables. Omitted from AWS CloudTrail logs.

Type: EnvironmentResponse object

**EphemeralStorage**

The size of the function's /tmp directory in MB. The default value is 512, but can be any whole number between 512 and 10,240 MB. For more information, see Configuring ephemeral storage (console).

Type: EphemeralStorage object

**FileSystemConfigs**

Connection settings for an Amazon EFS file system.

Type: Array of FileSystemConfig objects

Array Members: Maximum number of 1 item.

**FunctionArn**

The function's Amazon Resource Name (ARN).

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_.]+(:($LATEST|[a-zA-Z0-9-_.]+))?

**FunctionName**

The name of the function.

Type: String


Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}):(\d{12})?:?(function:)?([a-zA-Z0-9-_.]+):?(:($LATEST|[a-zA-Z0-9-_.]+))?

**Handler**

The function that Lambda calls to begin running your function.
Type: String

Length Constraints: Maximum length of 128.

Pattern: [^\s]+

**ImageConfigResponse**

The function's image configuration values.

Type: ImageConfigResponse object

**KMSKeyArn**

The AWS KMS key that's used to encrypt the function's environment variables. When Lambda SnapStart is activated, this key is also used to encrypt the function's snapshot. This key is returned only if you've configured a customer managed key.

Type: String

Pattern: (arn:(aws[a-zA-Z-]*)?:[a-z0-9-.*]+:[a-zA-Z0-9-]+:.*)()|

**LastModified**

The date and time that the function was last updated, in ISO-8601 format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

**LastUpdateStatus**

The status of the last update that was performed on the function. This is first set to Successful after function creation completes.

Type: String

Valid Values: Successful | Failed | InProgress

**LastUpdateStatusReason**

The reason for the last update that was performed on the function.

Type: String

**LastUpdateStatusReasonCode**

The reason code for the last update that was performed on the function.
Type: String


Layers

The function’s layers.

Type: Array of Layer objects

LoggingConfig

The function’s Amazon CloudWatch Logs configuration settings.

Type: LoggingConfig object

MasterArn

For Lambda@Edge functions, the ARN of the main function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-[a-z]+-[\d1]:\d{12}:function:[a-zA-Z0-9-_.]+(:($\{LATEST|[a-zA-Z0-9-_.]+))?

MemorySize

The amount of memory available to the function at runtime.

Type: Integer


PackageType

The type of deployment package. Set to Image for container image and set Zip for .zip file archive.

Type: String
Valid Values: Zip | Image

**RevisionId**

The latest updated revision of the function or alias.

Type: String

**Role**

The function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@-\_/]+`

**Runtime**

The identifier of the function's runtime. Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17 | ruby3.2 | python3.11 | nodejs20.x | provided.al2023 | python3.12 | java21

**RuntimeVersionConfig**

The ARN of the runtime and any errors that occurred.

Type: `RuntimeVersionConfig` object

**SigningJobArn**

The ARN of the signing job.

Type: String
AWS Lambda

Pattern: arn:(aws[a-zA-Z0-9-]*)(([a-zA-Z0-9\-]+\[a-zA-Z0-9\-]+\d\d|\d\d\d)\?\-[a-zA-Z]+(-gov)?\-[a-zA-Z]+\d}\d{1})?:\d{12}?:(.*)

**SigningProfileVersionArn**

The ARN of the signing profile version.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*)(([a-zA-Z0-9\-]+\[a-zA-Z0-9\-]+\d\d|\d\d\d)\?\-[a-zA-Z]+(-gov)?\-[a-zA-Z]+\d}\d{1})?:\d{12}?:(.*)

**SnapStart**

Set ApplyOn to PublishedVersions to create a snapshot of the initialized execution environment when you publish a function version. For more information, see [Improving startup performance with Lambda SnapStart](#).

Type: SnapStartResponse object

**State**

The current state of the function. When the state is Inactive, you can reactivate the function by invoking it.

Type: String

Valid Values: Pending | Active | Inactive | Failed

**StateReason**

The reason for the function's current state.

Type: String

**StateReasonCode**

The reason code for the function's current state. When the code is Creating, you can't invoke or modify the function.

Type: String

Valid Values: Idle | Creating | Restoring | EniLimitExceeded | InsufficientRolePermissions | InvalidConfiguration | InternalError
| SubnetOutOfIPAddresses | InvalidSubnet | InvalidSecurityGroup | ImageDeleted | ImageAccessDenied | InvalidImage | KMSKeyAccessDenied | KMSKeyNotFound | InvalidStateKMSKey | DisabledKMSKey | EFSIOError | EFSMountConnectivityError | EFSMountFailure | EFSMountTimeout | InvalidRuntime | InvalidZipFileException | FunctionError |

**Timeout**

The amount of time in seconds that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

**TracingConfig**

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse](#) object

**Version**

The version of the Lambda function.

Type: String


Pattern: `\$LATEST|\d+`

**VpcConfig**

The function's networking configuration.

Type: [VpcConfigResponse](#) object

**Errors**

**CodeStorageExceededException**

Your AWS account has exceeded its maximum total code size. For more information, see [Lambda quotas](#).

HTTP Status Code: 400
**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**PreconditionFailedException**

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

**ResourceConflictException**

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
• AWS SDK for C++
• AWS SDK for Go
• AWS SDK for Java V2
• AWS SDK for JavaScript V3
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
PutFunctionCodeSigningConfig

Update the code signing configuration for the function. Changes to the code signing configuration take effect the next time a user tries to deploy a code package to the function.

Request Syntax

```
PUT /2020-06-30/functions/FunctionName/code-signing-config HTTP/1.1
Content-type: application/json
{
   "CodeSigningConfigArn": "string"
}
```

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** - MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

**Length Constraints:** Minimum length of 1. Maximum length of 140.

```
Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-\_]+)(:\$LATEST|[a-zA-Z0-9-\_]+))?
```

**Required:** Yes

Request Body

The request accepts the following data in JSON format.
**CodeSigningConfigArn**

The Amazon Resource Name (ARN) of the code signing configuration.

Type: String

Length Constraints: Maximum length of 200.

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}((-gov)|(-iso(b?)))?-\[a-z]+-\d{1}:\d{12}:code-signing-config:csc-[a-z0-9]{17}`

Required: Yes

**Response Syntax**

```
HTTP/1.1 200
Content-type: application/json

{
   "CodeSigningConfigArn": "string",
   "FunctionName": "string"
}
```

**Response Elements**

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**CodeSigningConfigArn**

The Amazon Resource Name (ARN) of the code signing configuration.

Type: String

Length Constraints: Maximum length of 200.

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}((-gov)|(-iso(b?)))?-\[a-z]+-\d{1}:\d{12}:code-signing-config:csc-[a-z0-9]{17}`

**FunctionName**

The name of the Lambda function.
Name formats

- **Function name** - MyFunction.
- **Partial ARN** - `123456789012:function:MyFunction`.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-\_]+)(:\$LATEST|[a-zA-Z0-9-\_]+))?`

Errors

**CodeSigningConfigNotFoundException**

The specified code signing configuration does not exist.

HTTP Status Code: 404

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceConflictException**

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.
HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
PutFunctionConcurrency

Sets the maximum number of simultaneous executions for a function, and reserves capacity for that concurrency level.

Concurrency settings apply to the function as a whole, including all published versions and the unpublished version. Reserving concurrency both ensures that your function has capacity to process the specified number of events simultaneously, and prevents it from scaling beyond that level. Use `GetFunction` to see the current setting for a function.

Use `GetAccountSettings` to see your Regional concurrency limit. You can reserve concurrency for as many functions as you like, as long as you leave at least 100 simultaneous executions unreserved for functions that aren't configured with a per-function limit. For more information, see Lambda function scaling.

Request Syntax

```plaintext
PUT /2017-10-31/functions/FunctionName/concurrency HTTP/1.1
Content-type: application/json

{
    "ReservedConcurrentExecutions": number
}
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.
Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?\:lambda\:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?\d{12}:?function:)?([a-zA-Z0-9-\_]+)(:\$LATEST|\[a-zA-Z0-9-\_]+))?

Required: Yes

**Request Body**

The request accepts the following data in JSON format.

**ReservedConcurrentExecutions**

The number of simultaneous executions to reserve for the function.

Type: Integer

Valid Range: Minimum value of 0.

Required: Yes

**Response Syntax**

HTTP/1.1 200
Content-type: application/json

```
{
    "ReservedConcurrentExecutions": number
}
```

**Response Elements**

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**ReservedConcurrentExecutions**

The number of concurrent executions that are reserved for this function. For more information, see [Managing Lambda reserved concurrency](link).

PutFunctionConcurrency
Type: Integer

Valid Range: Minimum value of 0.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
AWS Lambda

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
PutFunctionEventInvokeConfig

Configures options for asynchronous invocation on a function, version, or alias. If a configuration already exists for a function, version, or alias, this operation overwrites it. If you exclude any settings, they are removed. To set one option without affecting existing settings for other options, use UpdateFunctionEventInvokeConfig.

By default, Lambda retries an asynchronous invocation twice if the function returns an error. It retains events in a queue for up to six hours. When an event fails all processing attempts or stays in the asynchronous invocation queue for too long, Lambda discards it. To retain discarded events, configure a dead-letter queue with UpdateFunctionConfiguration.

To send an invocation record to a queue, topic, function, or event bus, specify a destination. You can configure separate destinations for successful invocations (on-success) and events that fail all processing attempts (on-failure). You can configure destinations in addition to or instead of a dead-letter queue.

Request Syntax

```
PUT /2019-09-25/functions/FunctionName/event-invoke-config?Qualifier=Qualifier HTTP/1.1
Content-type: application/json

{
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "MaximumEventAgeInSeconds": number,
    "MaximumRetryAttempts": number
}
```

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function, version, or alias.
Name formats

- **Function name** - `my-function` (name-only), `my-function:v1` (with alias).
- **Partial ARN** - `123456789012:function:my-function`.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: \( \text{arn:aws[a-zA-Z-]*:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}):?function:)?([a-zA-Z0-9_-]+):?($LATEST|[a-zA-Z0-9_-]+)? \)

Required: Yes

**Qualifier**

A version number or alias name.


Pattern: \( |[a-zA-Z0-9-_]+ \)

**Request Body**

The request accepts the following data in JSON format.

**DestinationConfig**

A destination for events after they have been sent to a function for processing.

**Destinations**

- **Function** - The Amazon Resource Name (ARN) of a Lambda function.
- **Queue** - The ARN of a standard SQS queue.
- **Topic** - The ARN of a standard SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: **DestinationConfig** object

Required: No
**MaximumEventAgeInSeconds**

The maximum age of a request that Lambda sends to a function for processing.

Type: Integer

Valid Range: Minimum value of 60. Maximum value of 21600.

Required: No

**MaximumRetryAttempts**

The maximum number of times to retry when the function returns an error.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 2.

Required: No

**Response Syntax**

```
HTTP/1.1 200
Content-type: application/json

{
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "FunctionArn": "string",
    "LastModified": number,
    "MaximumEventAgeInSeconds": number,
    "MaximumRetryAttempts": number
}
```

**Response Elements**

If the action is successful, the service sends back an HTTP 200 response.
The following data is returned in JSON format by the service.

**DestinationConfig**

A destination for events after they have been sent to a function for processing.

**Destinations**

- **Function** - The Amazon Resource Name (ARN) of a Lambda function.
- **Queue** - The ARN of a standard SQS queue.
- **Topic** - The ARN of a standard SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: `DestinationConfig` object

**FunctionArn**

The Amazon Resource Name (ARN) of the function.

Type: String

Pattern: `arn:(aws[a-zA-Z]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-\_]+(:($LATEST|[a-zA-Z0-9-\_]+))?`

**LastModified**

The date and time that the configuration was last updated, in Unix time seconds.

Type: Timestamp

**MaximumEventAgeInSeconds**

The maximum age of a request that Lambda sends to a function for processing.

Type: Integer

Valid Range: Minimum value of 60. Maximum value of 21600.

**MaximumRetryAttempts**

The maximum number of times to retry when the function returns an error.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 2.
Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
**PutProvisionedConcurrencyConfig**

Adds a provisioned concurrency configuration to a function's alias or version.

**Request Syntax**

```plaintext
PUT /2019-09-30/functions/FunctionName/provisioned-concurrency?Qualifier=Qualifier
HTTP/1.1
Content-type: application/json

{
   "ProvisionedConcurrentExecutions": number
}
```

**URI Request Parameters**

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?(([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-]+):([\$LATEST|a-zA-Z0-9-]+))?

Required: Yes

**Qualifier**

The version number or alias name.

Pattern: (\[a-zA-Z0-9$_-]+)

Required: Yes

**Request Body**

The request accepts the following data in JSON format.

**ProvisionedConcurrentExecutions**

The amount of provisioned concurrency to allocate for the version or alias.

Type: Integer

Valid Range: Minimum value of 1.

Required: Yes

**Response Syntax**

HTTP/1.1 202
Content-type: application/json

```
{
   "AllocatedProvisionedConcurrentExecutions": number,
   "AvailableProvisionedConcurrentExecutions": number,
   "LastModified": "string",
   "RequestedProvisionedConcurrentExecutions": number,
   "Status": "string",
   "StatusReason": "string"
}
```

**Response Elements**

If the action is successful, the service sends back an HTTP 202 response.

The following data is returned in JSON format by the service.
**AllocatedProvisionedConcurrentExecutions**

The amount of provisioned concurrency allocated. When a weighted alias is used during linear and canary deployments, this value fluctuates depending on the amount of concurrency that is provisioned for the function versions.

Type: Integer

Valid Range: Minimum value of 0.

**AvailableProvisionedConcurrentExecutions**

The amount of provisioned concurrency available.

Type: Integer

Valid Range: Minimum value of 0.

**LastModified**

The date and time that a user last updated the configuration, in ISO 8601 format.

Type: String

**RequestedProvisionedConcurrentExecutions**

The amount of provisioned concurrency requested.

Type: Integer

Valid Range: Minimum value of 1.

**Status**

The status of the allocation process.

Type: String

Valid Values: IN_PROGRESS | READY | FAILED

**StatusReason**

For failed allocations, the reason that provisioned concurrency could not be allocated.

Type: String
Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
PutRuntimeManagementConfig

Sets the runtime management configuration for a function's version. For more information, see Runtime updates.

Request Syntax

PUT /2021-07-20/functions/FunctionName/runtime-management-config?Qualifier=Qualifier
HTTP/1.1
Content-type: application/json

{
    "RuntimeVersionArn": "string",
    "UpdateRuntimeOn": "string"
}

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-\_]*)?(:("LATEST|\d{12}:\d{12}:))?

Required: Yes
Qualifier

Specify a version of the function. This can be $LATEST or a published version number. If no value is specified, the configuration for the $LATEST version is returned.


Pattern: ([a-zA-Z0-9$_-]+)

Request Body

The request accepts the following data in JSON format.

RuntimeVersionArn

The ARN of the runtime version you want the function to use.

Note

This is only required if you're using the Manual runtime update mode.

Type: String


Pattern: ^arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}((-gov)|(-iso(b?)))?-[a-z]+-\d{1}:runtime:.+$

Required: No

UpdateRuntimeOn

Specify the runtime update mode.

- **Auto (default)** - Automatically update to the most recent and secure runtime version using a Two-phase runtime version rollout. This is the best choice for most customers to ensure they always benefit from runtime updates.

- **Function update** - Lambda updates the runtime of your function to the most recent and secure runtime version when you update your function. This approach synchronizes runtime
updates with function deployments, giving you control over when runtime updates are applied and allowing you to detect and mitigate rare runtime update incompatibilities early. When using this setting, you need to regularly update your functions to keep their runtime up-to-date.

- **Manual** - You specify a runtime version in your function configuration. The function will use this runtime version indefinitely. In the rare case where a new runtime version is incompatible with an existing function, this allows you to roll back your function to an earlier runtime version. For more information, see Roll back a runtime version.

  Type: String

  Valid Values: Auto | Manual | FunctionUpdate

  Required: Yes

### Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "FunctionArn": "string",
    "RuntimeVersionArn": "string",
    "UpdateRuntimeOn": "string"
}
```

### Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**FunctionArn**

The ARN of the function

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-]+:(\$LATEST|[a-zA-Z0-9-]+)\)`
**RuntimeVersionArn**

The ARN of the runtime the function is configured to use. If the runtime update mode is manual, the ARN is returned, otherwise null is returned.

Type: String


Pattern: `^arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}((-gov)|(-iso(b?)))?-[a-z]+\d{1}::runtime:.+$`

**UpdateRuntimeOn**

The runtime update mode.

Type: String

Valid Values: Auto | Manual | FunctionUpdate

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceConflictException**

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500
TooManyRequestsException

    The request throughput limit was exceeded. For more information, see Lambda quotas.

    HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
RemoveLayerVersionPermission

Removes a statement from the permissions policy for a version of an AWS Lambda layer. For more information, see AddLayerVersionPermission.

Request Syntax

DELETE /2018-10-31/layers/LayerName/versions/VersionNumber/policy/StatementId?RevisionId=RevisionId HTTP/1.1

URI Request Parameters

The request uses the following URI parameters.

LayerName

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-\_]+)|[a-zA-Z0-9-\_]+

Required: Yes

RevisionId

Only update the policy if the revision ID matches the ID specified. Use this option to avoid modifying a policy that has changed since you last read it.

StatementId

The identifier that was specified when the statement was added.

Length Constraints: Minimum length of 1. Maximum length of 100.

Pattern: ([a-zA-Z0-9-\_]+)

Required: Yes

VersionNumber

The version number.

Required: Yes
Request Body

The request does not have a request body.

Response Syntax

HTTP/1.1 204

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

PreconditionFailedException

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429
See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)
RemovePermission

Revokes function-use permission from an AWS service or another AWS account. You can get the ID of the statement from the output of `GetPolicy`.

Request Syntax

```
DELETE /2015-03-31/functions/FunctionName/policy/StatementId?
Qualifier=Qualifier&RevisionId=RevisionId HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function, version, or alias.

**Name formats**

- **Function name** – my-function (name-only), my-function:v1 (with alias).
- **Partial ARN** – 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]*)?(:($LATEST|[a-zA-Z0-9-_]*)+))?

Required: Yes

**Qualifier**

Specify a version or alias to remove permissions from a published version of the function.

Pattern: ( | [a-zA-Z0-9$-_.]+)

**RevisionId**

Update the policy only if the revision ID matches the ID that's specified. Use this option to avoid modifying a policy that has changed since you last read it.

**StatementId**

Statement ID of the permission to remove.

Length Constraints: Minimum length of 1. Maximum length of 100.

Pattern: ([a-zA-Z0-9-_.]+)

Required: Yes

**Request Body**

The request does not have a request body.

**Response Syntax**

HTTP/1.1 204

**Response Elements**

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**PreconditionFailedException**

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.
HTTP Status Code: 412

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](https://docs.aws.amazon.com/lambda/latest/dg/limits.html).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](https://aws.amazon.com/cli/)
- [AWS SDK for .NET](https://aws.amazon.com/sdk-for-net/)
- [AWS SDK for C++](https://aws.amazon.com/sdk-for-cpp/)
- [AWS SDK for Go](https://aws.amazon.com/sdk-for-go/)
- [AWS SDK for Java V2](https://aws.amazon.com/sdk-for-java-v2/)
- [AWS SDK for JavaScript V3](https://aws.amazon.com/sdk-for-javascript-v3/)
- [AWS SDK for PHP V3](https://aws.amazon.com/sdk-for-php-v3/)
- [AWS SDK for Python](https://aws.amazon.com/sdk-for-python/)
- [AWS SDK for Ruby V3](https://aws.amazon.com/sdk-for-ruby-v3/)
TagResource

Adds tags to a function.

Request Syntax

POST /2017-03-31/tags/ARN HTTP/1.1
Content-type: application/json

```json
{
    "Tags": {
        "string" : "string"
    }
}
```

URI Request Parameters

The request uses the following URI parameters.

**ARN**

The function's Amazon Resource Name (ARN).

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]\+))?

Required: Yes

Request Body

The request accepts the following data in JSON format.

**Tags**

A list of tags to apply to the function.

Type: String to string map

Required: Yes
Response Syntax

HTTP/1.1 204

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:
• AWS Command Line Interface
• AWS SDK for .NET
• AWS SDK for C++
• AWS SDK for Go
• AWS SDK for Java V2
• AWS SDK for JavaScript V3
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
UntagResource

Removes tags from a function.

Request Syntax

```
DELETE /2017-03-31/tags/ARN?tagKeys=TagKeys HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

**ARN**

The function's Amazon Resource Name (ARN).

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+--\d{1}:\d{12}:function:[a-zA-Z0-9-_.]+(:($LATEST|[a-zA-Z0-9-_.]+))?`

Required: Yes

**TagKeys**

A list of tag keys to remove from the function.

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.
Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
UpdateAlias

Updates the configuration of a Lambda function alias.

Request Syntax

PUT /2015-03-31/functions/{FunctionName}/aliases/{Name} HTTP/1.1
Content-type: application/json

{
    "Description": "string",
    "FunctionVersion": "string",
    "RevisionId": "string",
    "RoutingConfig": {
        "AdditionalVersionWeights": {
            "string": number
        }
    }
}

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-zA-Z]+-\d{1}):(\d{12}):(function:)?([a-zA-Z0-9-_.]+):($LATEST|[a-zA-Z0-9-_.]+)?
Required: Yes

Name

The name of the alias.


Pattern: (?![0-9]+$)([a-zA-Z0-9-]+)

Required: Yes

Request Body

The request accepts the following data in JSON format.

Description

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

FunctionVersion

The function version that the alias invokes.

Type: String


Pattern: ($LATEST|[0-9]+)

Required: No

RevisionId

Only update the alias if the revision ID matches the ID that's specified. Use this option to avoid modifying an alias that has changed since you last read it.

Type: String
Required: No

RoutingConfig

The routing configuration of the alias.

Type: AliasRoutingConfiguration object

Required: No

Response Syntax

HTTP/1.1 200
Content-type: application/json

```json
{
    "AliasArn": "string",
    "Description": "string",
    "FunctionVersion": "string",
    "Name": "string",
    "RevisionId": "string",
    "RoutingConfig": {
        "AdditionalVersionWeights": {
            "string": number
        }
    }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**AliasArn**

The Amazon Resource Name (ARN) of the alias.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]\+:\:($LATEST|[a-zA-Z0-9-_]\+)?
Description

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

FunctionVersion

The function version that the alias invokes.

Type: String


Pattern: (\$LATEST | [0-9]+)

Name

The name of the alias.

Type: String


Pattern: (?!^[0-9]+$)([a-zA-Z0-9-]+)

RevisionId

A unique identifier that changes when you update the alias.

Type: String

RoutingConfig

The routing configuration of the alias.

Type: AliasRoutingConfiguration object

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.
HTTP Status Code: 400

PreconditionFailedException

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
• AWS SDK for JavaScript V3
• AWS SDK for PHP V3
• AWS SDK for Python
• AWS SDK for Ruby V3
UpdateCodeSigningConfig

Update the code signing configuration. Changes to the code signing configuration take effect the next time a user tries to deploy a code package to the function.

Request Syntax

PUT /2020-04-22/code-signing-configs/CodeSigningConfigArn HTTP/1.1
Content-type: application/json

{
   "AllowedPublishers": {
      "SigningProfileVersionArns": [ "string" ]
   },
   "CodeSigningPolicies": {
      "UntrustedArtifactOnDeployment": "string"
   },
   "Description": "string"
}

URI Request Parameters

The request uses the following URI parameters.

**CodeSigningConfigArn**

The The Amazon Resource Name (ARN) of the code signing configuration.

Length Constraints: Maximum length of 200.

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z][2]((-gov)|(-iso(b?)))?-[a-z]+-\d{1}:\d{12}:code-signing-config:csc-[a-z0-9]{17}

Required: Yes

Request Body

The request accepts the following data in JSON format.

**AllowedPublishers**

Signing profiles for this code signing configuration.
Type: **AllowedPublishers** object

Required: No

**CodeSigningPolicies**

The code signing policy.

Type: **CodeSigningPolicies** object

Required: No

**Description**

Descriptive name for this code signing configuration.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

**Response Syntax**

```
HTTP/1.1 200
Content-type: application/json

{
    "CodeSigningConfig": {
        "AllowedPublishers": {
            "SigningProfileVersionArns": [ "string" ]
        },
        "CodeSigningConfigArn": "string",
        "CodeSigningConfigId": "string",
        "CodeSigningPolicies": {
            "UntrustedArtifactOnDeployment": "string"
        },
        "Description": "string",
        "LastModified": "string"
    }
}
```

**Response Elements**

If the action is successful, the service sends back an HTTP 200 response.
The following data is returned in JSON format by the service.

**CodeSigningConfig**

The code signing configuration

Type: `CodeSigningConfig` object

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
• AWS SDK for Python
• AWS SDK for Ruby V3
**UpdateEventSourceMapping**

Updates an event source mapping. You can change the function that AWS Lambda invokes, or pause invocation and resume later from the same location.

For details about how to configure different event sources, see the following topics.

- [Amazon DynamoDB Streams](#)
- [Amazon Kinesis](#)
- [Amazon SQS](#)
- [Amazon MQ and RabbitMQ](#)
- [Amazon MSK](#)
- [Apache Kafka](#)
- [Amazon DocumentDB](#)

The following error handling options are available only for stream sources (DynamoDB and Kinesis):

- **BisectBatchOnFunctionError** – If the function returns an error, split the batch in two and retry.
- **DestinationConfig** – Send discarded records to an Amazon SQS queue or Amazon SNS topic.
- **MaximumRecordAgeInSeconds** – Discard records older than the specified age. The default value is infinite (-1). When set to infinite (-1), failed records are retried until the record expires.
- **MaximumRetryAttempts** – Discard records after the specified number of retries. The default value is infinite (-1). When set to infinite (-1), failed records are retried until the record expires.
- **ParallelizationFactor** – Process multiple batches from each shard concurrently.

For information about which configuration parameters apply to each event source, see the following topics.

- [Amazon DynamoDB Streams](#)
- [Amazon Kinesis](#)
- [Amazon SQS](#)
- [Amazon MQ and RabbitMQ](#)
Request Syntax

PUT /2015-03-31/event-source-mappings/UUID HTTP/1.1
Content-type: application/json

{
    "BatchSize": number,
    "BisectBatchOnFunctionError": boolean,
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "DocumentDBEventSourceConfig": {
        "CollectionName": "string",
        "DatabaseName": "string",
        "FullDocument": "string"
    },
    "Enabled": boolean,
    "FilterCriteria": {
        "Filters": [
            {
                "Pattern": "string"
            }
        ]
    },
    "FunctionName": "string",
    "FunctionResponseTypes": [ "string" ],
    "MaximumBatchingWindowInSeconds": number,
    "MaximumRecordAgeInSeconds": number,
    "MaximumRetryAttempts": number,
    "ParallelizationFactor": number,
    "ScalingConfig": {
        "MaximumConcurrency": number
    }
}
"SourceAccessConfigurations": [
    {
        "Type": "string",
        "URI": "string"
    }
],
"TumblingWindowInSeconds": number
}

URI Request Parameters

The request uses the following URI parameters.

**UUID**

The identifier of the event source mapping.

Required: Yes

Request Body

The request accepts the following data in JSON format.

**BatchSize**

The maximum number of records in each batch that Lambda pulls from your stream or queue and sends to your function. Lambda passes all of the records in the batch to the function in a single call, up to the payload limit for synchronous invocation (6 MB).

- **Amazon Kinesis** – Default 100. Max 10,000.
- **Amazon DynamoDB Streams** – Default 100. Max 10,000.
- **Amazon Simple Queue Service** – Default 10. For standard queues the max is 10,000. For FIFO queues the max is 10.
- **Amazon Managed Streaming for Apache Kafka** – Default 100. Max 10,000.
- **Self-managed Apache Kafka** – Default 100. Max 10,000.
- **Amazon MQ (ActiveMQ and RabbitMQ)** – Default 100. Max 10,000.
- **DocumentDB** – Default 100. Max 10,000.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.
**BisectBatchOnFunctionError**

(Kinesis and DynamoDB Streams only) If the function returns an error, split the batch in two and retry.

Type: Boolean

Required: No

**DestinationConfig**

(Kinesis, DynamoDB Streams, Amazon MSK, and self-managed Kafka only) A configuration object that specifies the destination of an event after Lambda processes it.

Type: [DestinationConfig](#) object

Required: No

**DocumentDBEventSourceConfig**

Specific configuration settings for a DocumentDB event source.

Type: [DocumentDBEventSourceConfig](#) object

Required: No

**Enabled**

When true, the event source mapping is active. When false, Lambda pauses polling and invocation.

Default: True

Type: Boolean

Required: No

**FilterCriteria**

An object that defines the filter criteria that determine whether Lambda should process an event. For more information, see [Lambda event filtering](#).

Type: [FilterCriteria](#) object

Required: No
**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** – MyFunction.
- **Partial ARN** – 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it's limited to 64 characters in length.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?(([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?\d{12}:)?(function:)?([a-zA-Z0-9-\_]+)(:\$LATEST|[a-zA-Z0-9-\_]+))?

Required: No

**FunctionResponseTypes**

(Kinesis, DynamoDB Streams, and Amazon SQS) A list of current response type enums applied to the event source mapping.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 1 item.

Valid Values: ReportBatchItemFailures

Required: No

**MaximumBatchingWindowInSeconds**

The maximum amount of time, in seconds, that Lambda spends gathering records before invoking the function. You can configure MaximumBatchingWindowInSeconds to any value from 0 seconds to 300 seconds in increments of seconds.

For streams and Amazon SQS event sources, the default batching window is 0 seconds. For Amazon MSK, Self-managed Apache Kafka, Amazon MQ, and DocumentDB event
sources, the default batching window is 500 ms. Note that because you can only change MaximumBatchingWindowInSeconds in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

Related setting: For streams and Amazon SQS event sources, when you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

Required: No

**MaximumRecordAgeInSeconds**

(Kinesis and DynamoDB Streams only) Discard records older than the specified age. The default value is infinite (-1).

Type: Integer


Required: No

**MaximumRetryAttempts**

(Kinesis and DynamoDB Streams only) Discard records after the specified number of retries. The default value is infinite (-1). When set to infinite (-1), failed records are retried until the record expires.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

Required: No

**ParallelizationFactor**

(Kinesis and DynamoDB Streams only) The number of batches to process from each shard concurrently.

Type: Integer

**ScalingConfig**

(Amazon SQS only) The scaling configuration for the event source. For more information, see Configuring maximum concurrency for Amazon SQS event sources.

Type: `ScalingConfig` object

Required: No

**SourceAccessConfigurations**

An array of authentication protocols or VPC components required to secure your event source.

Type: Array of `SourceAccessConfiguration` objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

Required: No

**TumblingWindowInSeconds**

(Kinesis and DynamoDB Streams only) The duration in seconds of a processing window for DynamoDB and Kinesis Streams event sources. A value of 0 seconds indicates no tumbling window.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

Required: No

**Response Syntax**

```
HTTP/1.1 202
Content-type: application/json

{
   "AmazonManagedKafkaEventSourceConfig": {
      "ConsumerGroupId": "string"
   },
   "BatchSize": number,
   "BisectBatchOnFunctionError": boolean,
   "DestinationConfig": {

UpdateEventSourceMapping
```
"OnFailure": {
    "Destination": "string"
},
"OnSuccess": {
    "Destination": "string"
}
,"DocumentDBEventSourceConfig": {
    "CollectionName": "string",
    "DatabaseName": "string",
    "FullDocument": "string"
},
"EventSourceArn": "string",
"FilterCriteria": {
    "Filters": [
        {
            "Pattern": "string"
        }
    ]
},
"FunctionArn": "string",
"FunctionResponseTypes": [ "string" ],
"LastModified": number,
"LastProcessingResult": "string",
"MaximumBatchingWindowInSeconds": number,
"MaximumRecordAgeInSeconds": number,
"MaximumRetryAttempts": number,
"ParallelizationFactor": number,
"Queues": [ "string" ],
"ScalingConfig": {
    "MaximumConcurrency": number
},
"SelfManagedEventSource": {
    "Endpoints": {
        "string": [ "string" ]
    }
},
"SelfManagedKafkaEventSourceConfig": {
    "ConsumerGroupId": "string"
},
"SourceAccessConfigurations": [ {
    "Type": "string",
    "URI": "string"
Response Elements

If the action is successful, the service sends back an HTTP 202 response.

The following data is returned in JSON format by the service.

AmazonManagedKafkaEventSourceConfig

Specific configuration settings for an Amazon Managed Streaming for Apache Kafka (Amazon MSK) event source.

Type: AmazonManagedKafkaEventSourceConfig object

BatchSize

The maximum number of records in each batch that Lambda pulls from your stream or queue and sends to your function. Lambda passes all of the records in the batch to the function in a single call, up to the payload limit for synchronous invocation (6 MB).

Default value: Varies by service. For Amazon SQS, the default is 10. For all other services, the default is 100.

Related setting: When you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

BisectBatchOnFunctionError

(Kinesis and DynamoDB Streams only) If the function returns an error, split the batch in two and retry. The default value is false.
Type: Boolean

**DestinationConfig**

(Kinesis and DynamoDB Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: [DestinationConfig object](#)

**DocumentDBEventSourceConfig**

Specific configuration settings for a DocumentDB event source.

Type: [DocumentDBEventSourceConfig object](#)

**EventSourceArn**

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9-]+\d{1})?:\d{12}:function:[a-zA-Z0-9-]+(:\$LATEST|[a-zA-Z0-9-]+))?

**FilterCriteria**

An object that defines the filter criteria that determine whether Lambda should process an event. For more information, see [Lambda event filtering](#).

Type: [FilterCriteria object](#)

**FunctionArn**

The ARN of the Lambda function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*):lambda:[a-zA-Z0-9-]+([-gov]?-[a-zA-Z]+\d{1})?:\d{12}:function:[a-zA-Z0-9-]+(:\$LATEST|[a-zA-Z0-9-]+))?

**FunctionResponseTypes**

(Kinesis, DynamoDB Streams, and Amazon SQS) A list of current response type enums applied to the event source mapping.
Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 1 item.

Valid Values: ReportBatchItemFailures

**LastModified**

The date that the event source mapping was last updated or that its state changed, in Unix time seconds.

Type: Timestamp

**LastProcessingResult**

The result of the last Lambda invocation of your function.

Type: String

**MaximumBatchingWindowInSeconds**

The maximum amount of time, in seconds, that Lambda spends gathering records before invoking the function. You can configure MaximumBatchingWindowInSeconds to any value from 0 seconds to 300 seconds in increments of seconds.

For streams and Amazon SQS event sources, the default batching window is 0 seconds. For Amazon MSK, Self-managed Apache Kafka, Amazon MQ, and DocumentDB event sources, the default batching window is 500 ms. Note that because you can only change MaximumBatchingWindowInSeconds in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

Related setting: For streams and Amazon SQS event sources, when you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

**MaximumRecordAgeInSeconds**

(Kinesis and DynamoDB Streams only) Discard records older than the specified age. The default value is -1, which sets the maximum age to infinite. When the value is set to infinite, Lambda never discards old records.
Note

The minimum valid value for maximum record age is 60s. Although values less than 60 and greater than -1 fall within the parameter's absolute range, they are not allowed.

Type: Integer


**MaximumRetryAttempts**

(Kinesis and DynamoDB Streams only) Discard records after the specified number of retries. The default value is -1, which sets the maximum number of retries to infinite. When MaximumRetryAttempts is infinite, Lambda retries failed records until the record expires in the event source.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

**ParallelizationFactor**

(Kinesis and DynamoDB Streams only) The number of batches to process concurrently from each shard. The default value is 1.

Type: Integer


**Queues**

(Amazon MQ) The name of the Amazon MQ broker destination queue to consume.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 1000.

Pattern: \[\s\S\]*

**ScalingConfig**

(Amazon SQS only) The scaling configuration for the event source. For more information, see Configuring maximum concurrency for Amazon SQS event sources.
Type: `ScalingConfig` object

**SelfManagedEventSource**

The self-managed Apache Kafka cluster for your event source.

Type: `SelfManagedEventSource` object

**SelfManagedKafkaEventSourceConfig**

Specific configuration settings for a self-managed Apache Kafka event source.

Type: `SelfManagedKafkaEventSourceConfig` object

**SourceAccessConfigurations**

An array of the authentication protocol, VPC components, or virtual host to secure and define your event source.

Type: Array of `SourceAccessConfiguration` objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

**StartingPosition**

The position in a stream from which to start reading. Required for Amazon Kinesis and Amazon DynamoDB Stream event sources. AT_TIMESTAMP is supported only for Amazon Kinesis streams, Amazon DocumentDB, Amazon MSK, and self-managed Apache Kafka.

Type: String

Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

**StartingPositionTimestamp**

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds. StartingPositionTimestamp cannot be in the future.

Type: Timestamp

**State**

The state of the event source mapping. It can be one of the following: Creating, Enabling, Enabled, Disabling, Disabled, Updating, or Deleting.

Type: String
**StateTransitionReason**

Indicates whether a user or Lambda made the last change to the event source mapping.

Type: String

**Topics**

The name of the Kafka topic.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 249.

Pattern: `^[^.]([a-zA-Z0-9\-_.]+)+`

**TumblingWindowInSeconds**

(Kinesis and DynamoDB Streams only) The duration in seconds of a processing window for DynamoDB and Kinesis Streams event sources. A value of 0 seconds indicates no tumbling window.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

**UUID**

The identifier of the event source mapping.

Type: String

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceConflictException**

The resource already exists, or another operation is in progress.
HTTP Status Code: 409

**ResourceInUseException**

The operation conflicts with the resource's availability. For example, you tried to update an event source mapping in the CREATING state, or you tried to delete an event source mapping currently UPDATING.

HTTP Status Code: 400

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

### See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript V3](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
• AWS SDK for Ruby V3
UpdateFunctionCode

Updates a Lambda function's code. If code signing is enabled for the function, the code package must be signed by a trusted publisher. For more information, see Configuring code signing for Lambda.

If the function's package type is Image, then you must specify the code package in ImageUri as the URI of a container image in the Amazon ECR registry.

If the function's package type is Zip, then you must specify the deployment package as a .zip file archive. Enter the Amazon S3 bucket and key of the code .zip file location. You can also provide the function code inline using the ZipFile field.

The code in the deployment package must be compatible with the target instruction set architecture of the function (x86-64 or arm64).

The function's code is locked when you publish a version. You can't modify the code of a published version, only the unpublished version.

Note

For a function defined as a container image, Lambda resolves the image tag to an image digest. In Amazon ECR, if you update the image tag to a new image, Lambda does not automatically update the function.

Request Syntax

```
PUT /2015-03-31/functions/FunctionName/code HTTP/1.1
Content-type: application/json

{
    "Architectures": [ "string" ],
    "DryRun": boolean,
    "ImageUri": "string",
    "Publish": boolean,
    "RevisionId": "string",
    "S3Bucket": "string",
    "S3Key": "string",
    "S3ObjectVersion": "string",
    "ZipFile": blob
}
```
URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-9-]+)(:\($LATEST|[a-zA-Z0-9-9-]+\))?

Required: Yes

Request Body

The request accepts the following data in JSON format.

**Architectures**

The instruction set architecture that the function supports. Enter a string array with one of the valid values (arm64 or x86_64). The default value is x86_64.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: x86_64  |  arm64

Required: No
DryRun

Set to true to validate the request parameters and access permissions without modifying the function code.

Type: Boolean

Required: No

ImageUri

URI of a container image in the Amazon ECR registry. Do not use for a function defined with a .zip file archive.

Type: String

Required: No

Publish

Set to true to publish a new version of the function after updating the code. This has the same effect as calling PublishVersion separately.

Type: Boolean

Required: No

RevisionId

Update the function only if the revision ID matches the ID that’s specified. Use this option to avoid modifying a function that has changed since you last read it.

Type: String

Required: No

S3Bucket

An Amazon S3 bucket in the same AWS Region as your function. The bucket can be in a different AWS account. Use only with a function defined with a .zip file archive deployment package.

Type: String


Pattern: \^[0-9A-Za-z\.-_]*(?<!\.)$
Required: No

**S3Key**

The Amazon S3 key of the deployment package. Use only with a function defined with a .zip file archive deployment package.

Type: String


Required: No

**S3ObjectVersion**

For versioned objects, the version of the deployment package object to use.

Type: String


Required: No

**ZipFile**

The base64-encoded contents of the deployment package. AWS SDK and AWS CLI clients handle the encoding for you. Use only with a function defined with a .zip file archive deployment package.

Type: Base64-encoded binary data object

Required: No

**Response Syntax**

```json
HTTP/1.1 200
Content-type: application/json

{
    "Architectures": [ "string" ],
    "CodeSha256": "string",
    "CodeSize": number,
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
```
"Description": "string",
"Environment": {
    "Error": {
        "ErrorCode": "string",
        "Message": "string"
    },
    "Variables": {
        "string": "string"
    }
},
"EphemeralStorage": {
    "Size": number
},
"FileSystemConfigs": [
    {
        "Arn": "string",
        "LocalMountPath": "string"
    }
],
"FunctionArn": "string",
"FunctionName": "string",
"Handler": "string",
"ImageConfigResponse": {
    "Error": {
        "ErrorCode": "string",
        "Message": "string"
    },
    "ImageConfig": {
        "Command": [ "string" ],
        "EntryPoint": [ "string" ],
        "WorkingDirectory": "string"
    }
},
"KMSKeyArn": "string",
"LastModified": "string",
"LastUpdateStatus": "string",
"LastUpdateStatusReason": "string",
"LastUpdateStatusReasonCode": "string",
"Layers": [
    {
        "Arn": "string",
        "CodeSize": number,
        "SigningJobArn": "string",
        "SigningProfileVersionArn": "string"
    }
]
null

"LoggingConfig": {
  "ApplicationLogLevel": "string",
  "LogFormat": "string",
  "LogGroup": "string",
  "SystemLogLevel": "string"
},
"MasterArn": "string",
"MemorySize": number,
"PackageType": "string",
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"RuntimeVersionConfig": {
  "Error": {
    "ErrorCode": "string",
    "Message": "string"
  },
  "RuntimeVersionArn": "string"
},
"SigningJobArn": "string",
"SigningProfileVersionArn": "string",
"SnapStart": {
  "ApplyOn": "string",
  "OptimizationStatus": "string"
},
"State": "string",
"StateReason": "string",
"StateReasonCode": "string",
"Timeout": number,
"TracingConfig": {
  "Mode": "string"
},
"Version": "string",
"VpcConfig": {
  "Ipv6AllowedForDualStack": boolean,
  "SecurityGroupIds": [ "string" ],
  "SubnetIds": [ "string" ],
  "VpcId": "string"
}
Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Architectures

The instruction set architecture that the function supports. Architecture is a string array with one of the valid values. The default architecture value is `x86_64`.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: `x86_64` | `arm64`

CodeSha256

The SHA256 hash of the function's deployment package.

Type: String

CodeSize

The size of the function's deployment package, in bytes.

Type: Long

DeadLetterConfig

The function's dead letter queue.

Type: DeadLetterConfig object

Description

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Environment

The function's environment variables. Omitted from AWS CloudTrail logs.
**Type:** EnvironmentResponse object

**EphemeralStorage**

The size of the function's /tmp directory in MB. The default value is 512, but can be any whole number between 512 and 10,240 MB. For more information, see Configuring ephemeral storage (console).

**Type:** EphemeralStorage object

**FileSystemConfigs**

Connection settings for an Amazon EFS file system.

**Type:** Array of FileSystemConfig objects

Array Members: Maximum number of 1 item.

**FunctionArn**

The function's Amazon Resource Name (ARN).

**Type:** String

**Pattern:** arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-._]+(:(\$LATEST|[a-zA-Z0-9-._]+))?

**FunctionName**

The name of the function.

**Type:** String

**Length Constraints:** Minimum length of 1. Maximum length of 170.

**Pattern:** (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-._]+)(:(\$LATEST|[a-zA-Z0-9-._]+))?

**Handler**

The function that Lambda calls to begin running your function.

**Type:** String

**Length Constraints:** Maximum length of 128.
ImageConfigResponse

The function's image configuration values.

Type: ImageConfigResponse object

KMSKeyArn

The AWS KMS key that's used to encrypt the function's environment variables. When Lambda SnapStart is activated, this key is also used to encrypt the function's snapshot. This key is returned only if you've configured a customer managed key.

Type: String

Pattern: (arn:(aws[a-zA-Z-]*)?:[a-z0-9-.:]+:.*)|()

LastModified

The date and time that the function was last updated, in ISO-8601 format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

LastUpdateStatus

The status of the last update that was performed on the function. This is first set to Successful after function creation completes.

Type: String

Valid Values: Successful | Failed | InProgress

LastUpdateStatusReason

The reason for the last update that was performed on the function.

Type: String

LastUpdateStatusReasonCode

The reason code for the last update that was performed on the function.

Type: String

Layers

The function's layers.

Type: Array of Layer objects

LoggingConfig

The function's Amazon CloudWatch Logs configuration settings.

Type: LoggingConfig object

MasterArn

For Lambda@Edge functions, the ARN of the main function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]++(:($LATEST|[a-zA-Z0-9-_]++))?

MemorySize

The amount of memory available to the function at runtime.

Type: Integer


PackageType

The type of deployment package. Set to Image for container image and set Zip for .zip file archive.

Type: String

Valid Values: Zip | Image
**RevisionId**

The latest updated revision of the function or alias.

Type: String

**Role**

The function's execution role.

Type: String

Pattern: `arn:(aws[\-a-zA-Z]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\-\_\/]+`

**Runtime**

The identifier of the function's runtime. Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: String

Valid Values:
- nodejs
- nodejs4.3
- nodejs6.10
- nodejs8.10
- nodejs10.x
- nodejs12.x
- nodejs14.x
- nodejs16.x
- java8
- java8.al2
- java11
- python2.7
- python3.6
- python3.7
- python3.8
- python3.9
- dotnetcore1.0
- dotnetcore2.0
- dotnetcore2.1
- dotnetcore3.1
- dotnet6
- nodejs4.3-edge
- go1.x
- ruby2.5
- ruby2.7
- provided
- provided.al2
- nodejs18.x
- python3.10
- java17
- ruby3.2
- python3.11
- nodejs20.x
- provided.al2023
- python3.12
- java21

**RuntimeVersionConfig**

The ARN of the runtime and any errors that occurred.

Type: `RuntimeVersionConfig` object

**SigningJobArn**

The ARN of the signing job.

Type: String
**Pattern:** arn:(aws[a-zA-Z0-9-]*)::([a-zA-Z]{2}(-gov)?-[a-zA-Z]+\d{1})?:(\d{12})?:(.*)

**SigningProfileVersionArn**

The ARN of the signing profile version.

Type: String

**Pattern:** arn:(aws[a-zA-Z0-9-]*)::([a-zA-Z]{2}(-gov)?-[a-zA-Z]+\d{1})?:(\d{12})?:(.*)

**SnapStart**

Set ApplyOn to PublishedVersions to create a snapshot of the initialized execution environment when you publish a function version. For more information, see [Improving startup performance with Lambda SnapStart](#).

Type: SnapStartResponse object

**State**

The current state of the function. When the state is Inactive, you can reactivate the function by invoking it.

Type: String

Valid Values: Pending | Active | Inactive | Failed

**StateReason**

The reason for the function's current state.

Type: String

**StateReasonCode**

The reason code for the function's current state. When the code is Creating, you can't invoke or modify the function.

Type: String

Valid Values: Idle | Creating | Restoring | EniLimitExceeded | InsufficientRolePermissions | InvalidConfiguration | InternalError | SubnetOutOfIPAddresses | InvalidSubnet | InvalidSecurityGroup | ImageDeleted | ImageAccessDenied | InvalidImage | KMSKeyAccessDenied
Timeout

The amount of time in seconds that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

TracingConfig

The function's AWS X-Ray tracing configuration.

Type: TracingConfigResponse object

Version

The version of the Lambda function.

Type: String


Pattern: ($LATEST|[0-9]+)

VpcConfig

The function's networking configuration.

Type: VpcConfigResponse object

Errors

CodeSigningConfigNotFoundException

The specified code signing configuration does not exist.

HTTP Status Code: 404

CodeStorageExceededException

Your AWS account has exceeded its maximum total code size. For more information, see Lambda quotas.
HTTP Status Code: 400

**CodeVerificationFailedException**

The code signature failed one or more of the validation checks for signature mismatch or expiry, and the code signing policy is set to ENFORCE. Lambda blocks the deployment.

HTTP Status Code: 400

**InvalidCodeSignatureException**

The code signature failed the integrity check. If the integrity check fails, then Lambda blocks deployment, even if the code signing policy is set to WARN.

HTTP Status Code: 400

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**PreconditionFailedException**

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

**ResourceConflictException**

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500
TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
**UpdateFunctionConfiguration**

Modify the version-specific settings of a Lambda function.

When you update a function, Lambda provisions an instance of the function and its supporting resources. If your function connects to a VPC, this process can take a minute. During this time, you can't modify the function, but you can still invoke it. The `LastUpdateStatus`, `LastUpdateStatusReason`, and `LastUpdateStatusReasonCode` fields in the response from `GetFunctionConfiguration` indicate when the update is complete and the function is processing events with the new configuration. For more information, see [Lambda function states](#).

These settings can vary between versions of a function and are locked when you publish a version. You can't modify the configuration of a published version, only the unpublished version.

To configure function concurrency, use `PutFunctionConcurrency`. To grant invoke permissions to an AWS account or AWS service, use `AddPermission`.

**Request Syntax**

```
PUT /2015-03-31/functions/FunctionName/configuration HTTP/1.1
Content-type: application/json

{
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
    "Description": "string",
    "Environment": {
        "Variables": {
            "string": "string"
        }
    },
    "EphemeralStorage": {
        "Size": number
    },
    "FileSystemConfigs": [
        {
            "Arn": "string",
            "LocalMountPath": "string"
        }
    ],
    "Handler": "string",
    ...
}
```
"ImageConfig": {
  "Command": [ "string" ],
  "EntryPoint": [ "string" ],
  "WorkingDirectory": "string"
},
"KMSKeyArn": "string",
"Layers": [ "string" ],
"LoggingConfig": {
  "ApplicationLogLevel": "string",
  "LogFormat": "string",
  "LogGroup": "string",
  "SystemLogLevel": "string"
},
"MemorySize": number,
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"SnapStart": {
  "ApplyOn": "string"
},
"Timeout": number,
"TracingConfig": {
  "Mode": "string"
},
"VpcConfig": {
  "Ipv6AllowedForDualStack": boolean,
  "SecurityGroupIds": [ "string" ],
  "SubnetIds": [ "string" ]
}
}

**URI Request Parameters**

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function.

**Name formats**

- **Function name** – my-function.
• **Partial ARN** – 123456789012:function:my-function.

  The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

  Length Constraints: Minimum length of 1. Maximum length of 140.

  Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]*)((\$LATEST|[a-zA-Z0-9-_]++))?

  Required: Yes

**Request Body**

The request accepts the following data in JSON format.

**DeadLetterConfig**

A dead-letter queue configuration that specifies the queue or topic where Lambda sends asynchronous events when they fail processing. For more information, see [Dead-letter queues](#).  

Type: **DeadLetterConfig** object

Required: No

**Description**

A description of the function.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

**Environment**

Environment variables that are accessible from function code during execution.

Type: **Environment** object

Required: No
**EphemeralStorage**

The size of the function's /tmp directory in MB. The default value is 512, but can be any whole number between 512 and 10,240 MB. For more information, see Configuring ephemeral storage (console).

Type: EphemeralStorage object

Required: No

**FileSystemConfigs**

Connection settings for an Amazon EFS file system.

Type: Array of FileSystemConfig objects

Array Members: Maximum number of 1 item.

Required: No

**Handler**

The name of the method within your code that Lambda calls to run your function. Handler is required if the deployment package is a .zip file archive. The format includes the file name. It can also include namespaces and other qualifiers, depending on the runtime. For more information, see Lambda programming model.

Type: String

Length Constraints: Maximum length of 128.

Pattern: [\^\s]+

Required: No

**ImageConfig**

Container image configuration values that override the values in the container image Docker file.

Type: ImageConfig object

Required: No
**KMSKeyArn**

The ARN of the AWS Key Management Service (AWS KMS) customer managed key that's used to encrypt your function's environment variables. When Lambda SnapStart is activated, Lambda also uses this key to encrypt your function's snapshot. If you deploy your function using a container image, Lambda also uses this key to encrypt your function when it's deployed. Note that this is not the same key that's used to protect your container image in the Amazon Elastic Container Registry (Amazon ECR). If you don't provide a customer managed key, Lambda uses a default service key.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-z0-9-.:]+:)[a-zA-Z0-9-.:]+)`

Required: No

**Layers**

A list of function layers to add to the function's execution environment. Specify each layer by its ARN, including the version.

Type: Array of strings

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_:]+:[0-9]+`

Required: No

**LoggingConfig**

The function's Amazon CloudWatch Logs configuration settings.

Type: LoggingConfig object

Required: No

**MemorySize**

The amount of memory available to the function at runtime. Increasing the function memory also increases its CPU allocation. The default value is 128 MB. The value can be any multiple of 1 MB.

Type: Integer

Required: No

**RevisionId**

Update the function only if the revision ID matches the ID that's specified. Use this option to avoid modifying a function that has changed since you last read it.

Type: String

Required: No

**Role**

The Amazon Resource Name (ARN) of the function's execution role.

Type: String

Required: No

**Runtime**

The identifier of the function's runtime. Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see Runtime deprecation policy.

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17 | ruby3.2 | python3.11 | nodejs20.x | provided.al2023 | python3.12 | java21

Required: No

**SnapStart**

The function's SnapStart setting.
Type: SnapStart object

Required: No

**Timeout**

The amount of time (in seconds) that Lambda allows a function to run before stopping it. The default is 3 seconds. The maximum allowed value is 900 seconds. For more information, see Lambda execution environment.

Type: Integer

Valid Range: Minimum value of 1.

Required: No

**TracingConfig**

Set Mode to Active to sample and trace a subset of incoming requests with X-Ray.

Type: TracingConfig object

Required: No

**VpcConfig**

For network connectivity to AWS resources in a VPC, specify a list of security groups and subnets in the VPC. When you connect a function to a VPC, it can access resources and the internet only through that VPC. For more information, see Configuring a Lambda function to access resources in a VPC.

Type: VpcConfig object

Required: No

**Response Syntax**

```
HTTP/1.1 200
Content-type: application/json

{
   "Architectures": [ "string" ],
   "CodeSha256": "string",
   "CodeSize": number,
   "DeadLetterConfig": {

```
"TargetArn": "string",
"Description": "string",
"Environment": {
  "Error": {
    "ErrorCode": "string",
    "Message": "string"
  },
  "Variables": {
    "string": "string"
  }
},
"EphemeralStorage": {
  "Size": number
},
"FileSystemConfigs": [
  {
    "Arn": "string",
    "LocalMountPath": "string"
  }
],
"FunctionArn": "string",
"FunctionName": "string",
"Handler": "string",
"ImageConfigResponse": {
  "Error": {
    "ErrorCode": "string",
    "Message": "string"
  },
  "ImageConfig": {
    "Command": [ "string" ],
    "EntryPoint": [ "string" ],
    "WorkingDirectory": "string"
  }
},
"KMSKeyArn": "string",
"LastModified": "string",
"LastUpdateStatus": "string",
"LastUpdateStatusReason": "string",
"LastUpdateStatusReasonCode": "string",
"Layers": [
  {
    "Arn": "string",
    "CodeSize": number,
"SigningJobArn": "string",
"SigningProfileVersionArn": "string"
}
],
"LoggingConfig": {
  "ApplicationLogLevel": "string",
  "LogFormat": "string",
  "LogGroup": "string",
  "SystemLogLevel": "string"
},
"MasterArn": "string",
"MemorySize": number,
"PackageType": "string",
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"RuntimeVersionConfig": {
  "Error": {
    "ErrorCode": "string",
    "Message": "string"
  },
  "RuntimeVersionArn": "string"
},
"SigningJobArn": "string",
"SigningProfileVersionArn": "string",
"SnapStart": {
  "ApplyOn": "string",
  "OptimizationStatus": "string"
},
"State": "string",
"StateReason": "string",
"StateReasonCode": "string",
"Timeout": number,
"TracingConfig": {
  "Mode": "string"
},
"Version": "string",
"VpcConfig": {
  "Ipv6AllowedForDualStack": boolean,
  "SecurityGroupIds": [ "string" ],
  "SubnetIds": [ "string" ],
  "VpcId": "string"
}
Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Architectures

The instruction set architecture that the function supports. Architecture is a string array with one of the valid values. The default architecture value is x86_64.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: x86_64 | arm64

CodeSha256

The SHA256 hash of the function's deployment package.

Type: String

CodeSize

The size of the function's deployment package, in bytes.

Type: Long

DeadLetterConfig

The function's dead letter queue.

Type: DeadLetterConfig object

Description

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.
Environment

The function's environment variables. Omitted from AWS CloudTrail logs.

Type: EnvironmentResponse object

EphemeralStorage

The size of the function's /tmp directory in MB. The default value is 512, but can be any whole number between 512 and 10,240 MB. For more information, see Configuring ephemeral storage (console).

Type: EphemeralStorage object

FileSystemConfigs

Connection settings for an Amazon EFS file system.

Type: Array of FileSystemConfig objects

Array Members: Maximum number of 1 item.

FunctionArn

The function's Amazon Resource Name (ARN).

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:($LATEST|[a-zA-Z0-9-_\.]+))?

FunctionName

The name of the function.

Type: String


Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\$LATEST|[a-zA-Z0-9-_\.]+))?

Handler

The function that Lambda calls to begin running your function.
Type: String
Length Constraints: Maximum length of 128.
Pattern: [^\s]+

**ImageConfigResponse**

The function's image configuration values.

Type: `ImageConfigResponse` object

**KMSKeyArn**

The AWS KMS key that's used to encrypt the function's environment variables. When Lambda SnapStart is activated, this key is also used to encrypt the function's snapshot. This key is returned only if you've configured a customer managed key.

Type: String
Pattern: (arn:(aws[a-zA-Z-]*)?:[a-z0-9-.:]+.*)()|

**LastModified**

The date and time that the function was last updated, in ISO-8601 format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

**LastUpdateStatus**

The status of the last update that was performed on the function. This is first set to Successful after function creation completes.

Type: String
Valid Values: Successful | Failed | InProgress

**LastUpdateStatusReason**

The reason for the last update that was performed on the function.

Type: String

**LastUpdateStatusReasonCode**

The reason code for the last update that was performed on the function.
Type: String


**Layers**

The function's layers.

Type: Array of Layer objects

**LoggingConfig**

The function's Amazon CloudWatch Logs configuration settings.

Type: LoggingConfig object

**MasterArn**

For Lambda@Edge functions, the ARN of the main function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-]+(:($LATEST|[a-zA-Z0-9-]+))?

**MemorySize**

The amount of memory available to the function at runtime.

Type: Integer


**PackageType**

The type of deployment package. Set to Image for container image and set Zip for .zip file archive.

Type: String
Valid Values: Zip | Image

**RevisionId**

The latest updated revision of the function or alias.

Type: String

**Role**

The function's execution role.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\-_]+

**Runtime**

The identifier of the function's runtime. Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see Runtime deprecation policy.

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17 | ruby3.2 | python3.11 | nodejs20.x | provided.al2023 | python3.12 | java21

**RuntimeVersionConfig**

The ARN of the runtime and any errors that occurred.

Type: RuntimeVersionConfig object

**SigningJobArn**

The ARN of the signing job.

Type: String
Pattern: arn:(aws[a-zA-Z0-9-]*)(:([a-zA-Z0-9-])+:([a-z]{2}(-gov)?-[a-z]+-\d{1})?:(\d{12})?:(.*)

**SigningProfileVersionArn**

The ARN of the signing profile version.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*)(:([a-zA-Z0-9-])+:([a-z]{2}(-gov)?-[a-z]+-\d{1})?:(\d{12})?:(.*)

**SnapStart**

Set ApplyOn to PublishedVersions to create a snapshot of the initialized execution environment when you publish a function version. For more information, see Improving startup performance with Lambda SnapStart.

Type: SnapStartResponse object

**State**

The current state of the function. When the state is Inactive, you can reactivate the function by invoking it.

Type: String

Valid Values: Pending | Active | Inactive | Failed

**StateReason**

The reason for the function's current state.

Type: String

**StateReasonCode**

The reason code for the function's current state. When the code is Creating, you can't invoke or modify the function.

Type: String

Valid Values: Idle | Creating | Restoring | EniLimitExceeded | InsufficientRolePermissions | InvalidConfiguration | InternalError | SubnetOutOfIPAddresses | InvalidSubnet | InvalidSecurityGroup | ImageDeleted | ImageAccessDenied | InvalidImage | KMSKeyAccessDenied
The amount of time in seconds that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

The function's AWS X-Ray tracing configuration.

Type: TracingConfigResponse object

The version of the Lambda function.

Type: String


Pattern: (\$LATEST|[0-9]+)

The function's networking configuration.

Type: VpcConfigResponse object

The specified code signing configuration does not exist.

HTTP Status Code: 404

The code signature failed one or more of the validation checks for signature mismatch or expiry, and the code signing policy is set to ENFORCE. Lambda blocks the deployment.
HTTP Status Code: 400

InvalidCodeSignatureException

The code signature failed the integrity check. If the integrity check fails, then Lambda blocks deployment, even if the code signing policy is set to WARN.

HTTP Status Code: 400

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

PreconditionFailedException

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429
See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
UpdateFunctionEventInvokeConfig

Updates the configuration for asynchronous invocation for a function, version, or alias.

To configure options for asynchronous invocation, use `PutFunctionEventInvokeConfig`.

Request Syntax

```
POST /2019-09-25/functions/FunctionName/event-invoke-config?Qualifier=Qualifier
HTTP/1.1
Content-type: application/json

{
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "MaximumEventAgeInSeconds": number,
    "MaximumRetryAttempts": number
}
```

URI Request Parameters

The request uses the following URI parameters.

**FunctionName**

The name of the Lambda function, version, or alias.

Name formats

- **Function name** - `my-function` (name-only), `my-function:v1` (with alias).
- **Partial ARN** - `123456789012:function:my-function`.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.
Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z-A-Z0-9-_]*)?(($LATEST|[a-zA-Z0-9-_]*)+)?

Required: Yes

**Qualifier**

A version number or alias name.


Pattern: ([a-zA-Z0-9$_-]+)

**Request Body**

The request accepts the following data in JSON format.

**DestinationConfig**

A destination for events after they have been sent to a function for processing.

**Destinations**

- **Function** - The Amazon Resource Name (ARN) of a Lambda function.
- **Queue** - The ARN of a standard SQS queue.
- **Topic** - The ARN of a standard SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: **DestinationConfig** object

Required: No

**MaximumEventAgeInSeconds**

The maximum age of a request that Lambda sends to a function for processing.

Type: Integer

Valid Range: Minimum value of 60. Maximum value of 21600.

Required: No
**MaximumRetryAttempts**

The maximum number of times to retry when the function returns an error.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 2.

Required: No

**Response Syntax**

```
HTTP/1.1 200
Content-type: application/json

{
   "DestinationConfig": {
      "OnFailure": {
         "Destination": "string"
      },
      "OnSuccess": {
         "Destination": "string"
      }
   },
   "FunctionArn": "string",
   "LastModified": number,
   "MaximumEventAgeInSeconds": number,
   "MaximumRetryAttempts": number
}
```

**Response Elements**

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**DestinationConfig**

A destination for events after they have been sent to a function for processing.

**Destinations**

- **Function** - The Amazon Resource Name (ARN) of a Lambda function.
- **Queue** - The ARN of a standard SQS queue.
- **Topic** - The ARN of a standard SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: `DestinationConfig` object

**FunctionArn**

The Amazon Resource Name (ARN) of the function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]++:\$LATEST[a-zA-Z0-9-_]++`?

**LastModified**

The date and time that the configuration was last updated, in Unix time seconds.

Type: Timestamp

**MaximumEventAgeInSeconds**

The maximum age of a request that Lambda sends to a function for processing.

Type: Integer

Valid Range: Minimum value of 60. Maximum value of 21600.

**MaximumRetryAttempts**

The maximum number of times to retry when the function returns an error.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 2.

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400
ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for JavaScript V3
- AWS SDK for PHP V3
- AWS SDK for Python
- AWS SDK for Ruby V3
UpdateFunctionUrlConfig

Updates the configuration for a Lambda function URL.

Request Syntax

PUT /2021-10-31/functions/FunctionName/url?Qualifier=Qualifier HTTP/1.1
Content-type: application/json

{
   "AuthType": "string",
   "Cors": {
      "AllowCredentials": boolean,
      "AllowHeaders": [ "string" ],
      "AllowMethods": [ "string" ],
      "AllowOrigins": [ "string" ],
      "ExposeHeaders": [ "string" ],
      "MaxAge": number
   },
   "InvokeMode": "string"
}

URI Request Parameters

The request uses the following URI parameters.

FunctionName

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.
Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-\_]\+)(:($LATEST|[a-zA-Z0-9-\_]+))?

Required: Yes

**Qualifier**

The alias name.


Pattern: (%LATEST%)|((?!^[0-9]+$)([a-zA-Z0-9-\_]+))

**Request Body**

The request accepts the following data in JSON format.

**AuthType**

The type of authentication that your function URL uses. Set to AWS_IAM if you want to restrict access to authenticated users only. Set to NONE if you want to bypass IAM authentication to create a public endpoint. For more information, see Security and auth model for Lambda function URLs.

Type: String

Valid Values: NONE | AWS_IAM

Required: No

**Cors**

The cross-origin resource sharing (CORS) settings for your function URL.

Type: Cors object

Required: No

**InvokeMode**

Use one of the following options:

- BUFFERED – This is the default option. Lambda invokes your function using the Invoke API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.
AWS Lambda Developer Guide

- **RESPONSE_STREAM** – Your function streams payload results as they become available. Lambda invokes your function using the `InvokeWithResponseStream` API operation. The maximum response payload size is 20 MB, however, you can request a quota increase.

  Type: String

  Valid Values: BUFFERED | RESPONSE_STREAM

  Required: No

**Response Syntax**

HTTP/1.1 200
Content-type: application/json

{  
  "AuthType": "string",
  "Cors": {
    "AllowCredentials": boolean,
    "AllowHeaders": [ "string" ],
    "AllowMethods": [ "string" ],
    "AllowOrigins": [ "string" ],
    "ExposeHeaders": [ "string" ],
    "MaxAge": number
  },
  "CreationTime": "string",
  "FunctionArn": "string",
  "FunctionUrl": "string",
  "InvokeMode": "string",
  "LastModifiedTime": "string"
}

**Response Elements**

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

**AuthType**

The type of authentication that your function URL uses. Set to AWS_IAM if you want to restrict access to authenticated users only. Set to NONE if you want to bypass IAM authentication to
create a public endpoint. For more information, see [Security and auth model for Lambda function URLs](#).

Type: String

Valid Values: NONE | AWS_IAM

**Cors**

The cross-origin resource sharing (CORS) settings for your function URL.

Type: Cors object

**CreationTime**

When the function URL was created, in [ISO-8601 format](https://en.wikipedia.org/wiki/ISO_8601) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

**FunctionArn**

The Amazon Resource Name (ARN) of your function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-[a-zA-Z0-9-_:]+(-\$LATEST|[a-zA-Z0-9-_:]+)?

**FunctionUrl**

The HTTP URL endpoint for your function.

Type: String

Length Constraints: Minimum length of 40. Maximum length of 100.

**InvokeMode**

Use one of the following options:

- **BUFFERED** – This is the default option. Lambda invokes your function using the Invoke API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.

- **RESPONSE_STREAM** – Your function streams payload results as they become available. Lambda invokes your function using the InvokeWithResponseStream API operation. The maximum response payload size is 20 MB, however, you can request a quota increase.

Type: String
Valid Values: BUFFERED | RESPONSE_STREAM

**LastModifiedTime**

When the function URL configuration was last updated, in **ISO-8601 format** (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

**Errors**

**InvalidParameterValueException**

One of the parameters in the request is not valid.

HTTP Status Code: 400

**ResourceConflictException**

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

**ResourceNotFoundException**

The resource specified in the request does not exist.

HTTP Status Code: 404

**ServiceException**

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

**TooManyRequestsException**

The request throughput limit was exceeded. For more information, see Lambda quotas.

HTTP Status Code: 429

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:
Data Types

The following data types are supported:

- AccountLimit
- AccountUsage
- AliasConfiguration
- AliasRoutingConfiguration
- AllowedPublishers
- AmazonManagedKafkaEventSourceConfig
- CodeSigningConfig
- CodeSigningPolicies
- Concurrency
- Cors
- DeadLetterConfig
- DestinationConfig
- DocumentDBEventSourceConfig
- Environment
- EnvironmentError
- EnvironmentResponse
- EphemeralStorage
- EventSourceMappingConfiguration
- FileSystemConfig
- Filter
- FilterCriteria
- FunctionCode
- FunctionCodeLocation
- FunctionConfiguration
- FunctionEventInvokeConfig
- FunctionUrlConfig
- ImageConfig
- ImageConfigError
- ImageConfigResponse
- InvokeResponseStreamUpdate
- InvokeWithResponseStreamCompleteEvent
- InvokeWithResponseStreamResponseEvent
- Layer
- LayersListItem
- LayerVersionContentInput
- LayerVersionContentOutput
- LayerVersionsListItem
- LoggingConfig
- OnFailure
- OnSuccess
- ProvisionedConcurrencyConfigListItem
- RuntimeVersionConfig
- RuntimeVersionError
- ScalingConfig
- SelfManagedEventSource
- SelfManagedKafkaEventSourceConfig
- SnapStart
- SnapStartResponse
- SourceAccessConfiguration
- TracingConfig
- TracingConfigResponse
- VpcConfig
- VpcConfigResponse
AccountLimit

Limits that are related to concurrency and storage. All file and storage sizes are in bytes.

Contents

CodeSizeUnzipped

The maximum size of a function's deployment package and layers when they're extracted.

Type: Long

Required: No

CodeSizeZipped

The maximum size of a deployment package when it's uploaded directly to Lambda. Use Amazon S3 for larger files.

Type: Long

Required: No

ConcurrentExecutions

The maximum number of simultaneous function executions.

Type: Integer

Required: No

TotalCodeSize

The amount of storage space that you can use for all deployment packages and layer archives.

Type: Long

Required: No

UnreservedConcurrentExecutions

The maximum number of simultaneous function executions, minus the capacity that's reserved for individual functions with PutFunctionConcurrency.

Type: Integer
Valid Range: Minimum value of 0.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
AccountUsage

The number of functions and amount of storage in use.

Contents

FunctionCount

The number of Lambda functions.

Type: Long

Required: No

TotalCodeSize

The amount of storage space, in bytes, that's being used by deployment packages and layer archives.

Type: Long

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
AliasConfiguration

Provides configuration information about a Lambda function alias.

Contents

AliasArn

The Amazon Resource Name (ARN) of the alias.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+?-\d{1}:
\d{12}:function:[a-zA-Z0-9-_.]+(:($LATEST|[a-zA-Z0-9-_.]+))?

Required: No

Description

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

FunctionVersion

The function version that the alias invokes.

Type: String


Pattern: ($LATEST|[0-9]+)

Required: No

Name

The name of the alias.

Type: String

Pattern: (?!^[0-9]+$)([a-zA-Z0-9-_.]+)

Required: No

**RevisionId**

A unique identifier that changes when you update the alias.

Type: String

Required: No

**RoutingConfig**

The routing configuration of the alias.

Type: `AliasRoutingConfiguration` object

Required: No

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
AliasRoutingConfiguration

The traffic-shifting configuration of a Lambda function alias.

Contents

AdditionalVersionWeights

The second version, and the percentage of traffic that's routed to it.

Type: String to double map

Key Length Constraints: Minimum length of 1. Maximum length of 1024.

Key Pattern: [0-9]+

Valid Range: Minimum value of 0.0. Maximum value of 1.0.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
AllowedPublishers

List of signing profiles that can sign a code package.

Contents

SigningProfileVersionArns

The Amazon Resource Name (ARN) for each of the signing profiles. A signing profile defines a trusted user who can sign a code package.

Type: Array of strings

Array Members: Minimum number of 1 item. Maximum number of 20 items.

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+:([a-z]{2}(-gov)?-[a-z]+-\d{1})?:\d{12}?:(.*)

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
AmazonManagedKafkaEventSourceConfig

Specific configuration settings for an Amazon Managed Streaming for Apache Kafka (Amazon MSK) event source.

Contents

ConsumerGroupId

The identifier for the Kafka consumer group to join. The consumer group ID must be unique among all your Kafka event sources. After creating a Kafka event source mapping with the consumer group ID specified, you cannot update this value. For more information, see Customizable consumer group ID.

Type: String


Pattern: [a-zA-Z0-9-_\/*:+=.@-]*

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
**CodeSigningConfig**


**Contents**

**AllowedPublishers**

List of allowed publishers.

Type: [AllowedPublishers](#) object

Required: Yes

**CodeSigningConfigArn**

The Amazon Resource Name (ARN) of the Code signing configuration.

Type: String

Length Constraints: Maximum length of 200.

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}((-gov)|(iso(b?)))?-[a-z]+\d{1}:\d{12}:code-signing-config:csc-[a-z0-9]{17}`

Required: Yes

**CodeSigningConfigId**

Unique identifier for the Code signing configuration.

Type: String

Pattern: `csc-[a-zA-Z0-9-_.]{17}`

Required: Yes

**CodeSigningPolicies**

The code signing policy controls the validation failure action for signature mismatch or expiry.

Type: [CodeSigningPolicies](#) object

Required: Yes
**LastModified**

The date and time that the Code signing configuration was last modified, in ISO-8601 format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Required: Yes

**Description**

Code signing configuration description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
CodeSigningPolicies

Code signing configuration policies specify the validation failure action for signature mismatch or expiry.

Contents

UntrustedArtifactOnDeployment

Code signing configuration policy for deployment validation failure. If you set the policy to Enforce, Lambda blocks the deployment request if signature validation checks fail. If you set the policy to Warn, Lambda allows the deployment and creates a CloudWatch log.

Default value: Warn

Type: String

Valid Values: Warn | Enforce

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
Concurrency

Contents

ReservedConcurrentExecutions

The number of concurrent executions that are reserved for this function. For more information, see Managing Lambda reserved concurrency.

Type: Integer

Valid Range: Minimum value of 0.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
Cors

The cross-origin resource sharing (CORS) settings for your Lambda function URL. Use CORS to grant access to your function URL from any origin. You can also use CORS to control access for specific HTTP headers and methods in requests to your function URL.

Contents

AllowCredentials

Whether to allow cookies or other credentials in requests to your function URL. The default is false.

Type: Boolean

Required: No

AllowHeaders

The HTTP headers that origins can include in requests to your function URL. For example: Date, Keep-Alive, X-Custom-Header.

Type: Array of strings

Array Members: Maximum number of 100 items.

Length Constraints: Maximum length of 1024.

Pattern: .*

Required: No

AllowMethods

The HTTP methods that are allowed when calling your function URL. For example: GET, POST, DELETE, or the wildcard character (*).

Type: Array of strings

Array Members: Maximum number of 6 items.

Length Constraints: Maximum length of 6.

Pattern: .*
**AllowOrigins**

The origins that can access your function URL. You can list any number of specific origins, separated by a comma. For example: https://www.example.com, http://localhost:60905.

Alternatively, you can grant access to all origins using the wildcard character (*).

Type: Array of strings

Array Members: Maximum number of 100 items.


Pattern: .*

Required: No

**ExposeHeaders**

The HTTP headers in your function response that you want to expose to origins that call your function URL. For example: Date, Keep-Alive, X-Custom-Header.

Type: Array of strings

Array Members: Maximum number of 100 items.

Length Constraints: Maximum length of 1024.

Pattern: .*

Required: No

**MaxAge**

The maximum amount of time, in seconds, that web browsers can cache results of a preflight request. By default, this is set to 0, which means that the browser doesn't cache results.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 86400.

Required: No
See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
DeadLetterConfig

The **dead-letter queue** for failed asynchronous invocations.

## Contents

### TargetArn

The Amazon Resource Name (ARN) of an Amazon SQS queue or Amazon SNS topic.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*):[a-z0-9-.]+:.*)|()`

Required: No

## See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
DestinationConfig

A configuration object that specifies the destination of an event after Lambda processes it.

Contents

OnFailure

The destination configuration for failed invocations.

Type: OnFailure object

Required: No

OnSuccess

The destination configuration for successful invocations.

Type: OnSuccess object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
DocumentDBEventSourceConfig

Specific configuration settings for a DocumentDB event source.

Contents

CollectionName

The name of the collection to consume within the database. If you do not specify a collection, Lambda consumes all collections.

Type: String


Pattern: (^(?!system\xe2))(^_[a-zA-Z0-9])([^$]*)

Required: No

DatabaseName

The name of the database to consume within the DocumentDB cluster.

Type: String


Pattern: [^ /\.\x22]*

Required: No

FullDocument

Determines what DocumentDB sends to your event stream during document update operations. If set to UpdateLookup, DocumentDB sends a delta describing the changes, along with a copy of the entire document. Otherwise, DocumentDB sends only a partial document that contains the changes.

Type: String

Valid Values: UpdateLookup | Default

Required: No
See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
Environment

A function's environment variable settings. You can use environment variables to adjust your function's behavior without updating code. An environment variable is a pair of strings that are stored in a function's version-specific configuration.

Contents

Variables

Environment variable key-value pairs. For more information, see Using Lambda environment variables.

Type: String to string map

Key Pattern: [a-zA-Z][a-zA-Z0-9_]++

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
EnvironmentError

Error messages for environment variables that couldn't be applied.

Contents

ErrorCode

The error code.

Type: String

Required: No

Message

The error message.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
EnvironmentResponse

The results of an operation to update or read environment variables. If the operation succeeds, the response contains the environment variables. If it fails, the response contains details about the error.

Contents

Error

Error messages for environment variables that couldn't be applied.

Type: EnvironmentError object

Required: No

Variables

Environment variable key-value pairs. Omitted from AWS CloudTrail logs.

Type: String to string map

Key Pattern: [a-zA-Z]( [a-zA-Z0-9_]+)

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
EphemeralStorage

The size of the function's `/tmp` directory in MB. The default value is 512, but can be any whole number between 512 and 10,240 MB. For more information, see Configuring ephemeral storage (console).

Contents

Size

The size of the function's `/tmp` directory.

Type: Integer


Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
EventSourceMappingConfiguration

A mapping between an AWS resource and a Lambda function. For details, see CreateEventSourceMapping.

Contents

AmazonManagedKafkaEventSourceConfig

Specific configuration settings for an Amazon Managed Streaming for Apache Kafka (Amazon MSK) event source.

Type: AmazonManagedKafkaEventSourceConfig object

Required: No

BatchSize

The maximum number of records in each batch that Lambda pulls from your stream or queue and sends to your function. Lambda passes all of the records in the batch to the function in a single call, up to the payload limit for synchronous invocation (6 MB).

Default value: Varies by service. For Amazon SQS, the default is 10. For all other services, the default is 100.

Related setting: When you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

Required: No

BisectBatchOnFunctionError

(Kinesis and DynamoDB Streams only) If the function returns an error, split the batch in two and retry. The default value is false.

Type: Boolean

Required: No
**DestinationConfig**

(Kinesis and DynamoDB Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: `DestinationConfig` object

Required: No

**DocumentDBEventSourceConfig**

Specific configuration settings for a DocumentDB event source.

Type: `DocumentDBEventSourceConfig` object

Required: No

**EventSourceArn**

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*):(a-zA-Z0-9-]+):([a-z]{2}(-gov)?-[a-z]+-\d{16}):(\d{12})?:(.*)`

Required: No

**FilterCriteria**

An object that defines the filter criteria that determine whether Lambda should process an event. For more information, see [Lambda event filtering](#).

Type: `FilterCriteria` object

Required: No

**FunctionArn**

The ARN of the Lambda function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:([a-z]{2}(-gov)?-[a-z]+-\d{16}):(\d{12}):function:[a-zA-Z0-9-]*:(.*)`

Required: No
FunctionResponseTypes

(Kinesis, DynamoDB Streams, and Amazon SQS) A list of current response type enums applied to the event source mapping.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 1 item.

Valid Values: ReportBatchItemFailures

Required: No

LastModified

The date that the event source mapping was last updated or that its state changed, in Unix time seconds.

Type: Timestamp

Required: No

LastProcessingResult

The result of the last Lambda invocation of your function.

Type: String

Required: No

MaximumBatchingWindowInSeconds

The maximum amount of time, in seconds, that Lambda spends gathering records before invoking the function. You can configure MaximumBatchingWindowInSeconds to any value from 0 seconds to 300 seconds in increments of seconds.

For streams and Amazon SQS event sources, the default batching window is 0 seconds.
For Amazon MSK, Self-managed Apache Kafka, Amazon MQ, and DocumentDB event sources, the default batching window is 500 ms. Note that because you can only change MaximumBatchingWindowInSeconds in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

Related setting: For streams and Amazon SQS event sources, when you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.
Type: Integer
Valid Range: Minimum value of 0. Maximum value of 300.
Required: No

**MaximumRecordAgeInSeconds**

(Kinesis and DynamoDB Streams only) Discard records older than the specified age. The default value is -1, which sets the maximum age to infinite. When the value is set to infinite, Lambda never discards old records.

**Note**
The minimum valid value for maximum record age is 60s. Although values less than 60 and greater than -1 fall within the parameter's absolute range, they are not allowed.

Type: Integer
Required: No

**MaximumRetryAttempts**

(Kinesis and DynamoDB Streams only) Discard records after the specified number of retries. The default value is -1, which sets the maximum number of retries to infinite. When MaximumRetryAttempts is infinite, Lambda retries failed records until the record expires in the event source.

Type: Integer
Valid Range: Minimum value of -1. Maximum value of 10000.
Required: No

**ParallelizationFactor**

(Kinesis and DynamoDB Streams only) The number of batches to process concurrently from each shard. The default value is 1.

Type: Integer

Required: No

Queues

(Amazon MQ) The name of the Amazon MQ broker destination queue to consume.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 1000.

Pattern: [\s\S]*

Required: No

ScalingConfig

(Amazon SQS only) The scaling configuration for the event source. For more information, see Configuring maximum concurrency for Amazon SQS event sources.

Type: ScalingConfig object

Required: No

SelfManagedEventSource

The self-managed Apache Kafka cluster for your event source.

Type: SelfManagedEventSource object

Required: No

SelfManagedKafkaEventSourceConfig

Specific configuration settings for a self-managed Apache Kafka event source.

Type: SelfManagedKafkaEventSourceConfig object

Required: No

SourceAccessConfigurations

An array of the authentication protocol, VPC components, or virtual host to secure and define your event source.
Type: Array of `SourceAccessConfiguration` objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

Required: No

**StartingPosition**

The position in a stream from which to start reading. Required for Amazon Kinesis and Amazon DynamoDB Stream event sources. AT_TIMESTAMP is supported only for Amazon Kinesis streams, Amazon DocumentDB, Amazon MSK, and self-managed Apache Kafka.

Type: String

Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

Required: No

**StartingPositionTimestamp**

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds. StartingPositionTimestamp cannot be in the future.

Type: Timestamp

Required: No

**State**

The state of the event source mapping. It can be one of the following: Creating, Enabling, Enabled, Disabling, Disabled, Updating, or Deleting.

Type: String

Required: No

**StateTransitionReason**

Indicates whether a user or Lambda made the last change to the event source mapping.

Type: String

Required: No

**Topics**

The name of the Kafka topic.
Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 249.

Pattern: ^[^.](\[a-zA-Z0-9\-_.]+)+

Required: No

**TumblingWindowInSeconds**

(Kinesis and DynamoDB Streams only) The duration in seconds of a processing window for DynamoDB and Kinesis Streams event sources. A value of 0 seconds indicates no tumbling window.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

Required: No

**UUID**

The identifier of the event source mapping.

Type: String

Required: No

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
FileSystemConfig

Details about the connection between a Lambda function and an Amazon EFS file system.

Contents

Arn

The Amazon Resource Name (ARN) of the Amazon EFS access point that provides access to the file system.

Type: String

Length Constraints: Maximum length of 200.

Pattern: arn:aws[a-zA-Z-A-Z-]*:elasticfilesystem:[a-z]{2}((-gov)|(-iso(b?)))?[a-z]+-\d[1]\:\d[12]:access-point/fsap-[a-f0-9]{17}

Required: Yes

LocalMountPath

The path where the function can access the file system, starting with /mnt/.

Type: String

Length Constraints: Maximum length of 160.

Pattern: ^/mnt/[a-zA-Z0-9-_.]+$

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
Filter

A structure within a FilterCriteria object that defines an event filtering pattern.

Contents

Pattern

A filter pattern. For more information on the syntax of a filter pattern, see Filter rule syntax.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 4096.

Pattern: .*

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
FilterCriteria

An object that contains the filters for an event source.

Contents

Filters

A list of filters.

Type: Array of Filter objects

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
**FunctionCode**

The code for the Lambda function. You can either specify an object in Amazon S3, upload a .zip file archive deployment package directly, or specify the URI of a container image.

**Contents**

**ImageUri**

URI of a [container image](#) in the Amazon ECR registry.

Type: String

Required: No

**S3Bucket**

An Amazon S3 bucket in the same AWS Region as your function. The bucket can be in a different AWS account.

Type: String


Pattern: ^[0-9A-Za-z\-\.\-_]*(?<!\.)$

Required: No

**S3Key**

The Amazon S3 key of the deployment package.

Type: String


Required: No

**S3ObjectVersion**

For versioned objects, the version of the deployment package object to use.

Type: String

Required: No

ZipFile

The base64-encoded contents of the deployment package. AWS SDK and AWS CLI clients handle the encoding for you.

Type: Base64-encoded binary data object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
FunctionCodeLocation

Details about a function's deployment package.

Contents

ImageUri

URI of a container image in the Amazon ECR registry.

Type: String

Required: No

Location

A presigned URL that you can use to download the deployment package.

Type: String

Required: No

RepositoryType

The service that's hosting the file.

Type: String

Required: No

ResolvedImageUri

The resolved URI for the image.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- **AWS SDK for Go**
- **AWS SDK for Java V2**
- **AWS SDK for Ruby V3**
FunctionConfiguration

Details about a function's configuration.

Contents

Architectures

The instruction set architecture that the function supports. Architecture is a string array with one of the valid values. The default architecture value is x86_64.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: x86_64 | arm64

Required: No

CodeSha256

The SHA256 hash of the function's deployment package.

Type: String

Required: No

CodeSize

The size of the function's deployment package, in bytes.

Type: Long

Required: No

DeadLetterConfig

The function's dead letter queue.

Type: DeadLetterConfig object

Required: No

Description

The function's description.
AWS Lambda

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

**Environment**

The function's environment variables. Omitted from AWS CloudTrail logs.

Type: EnvironmentResponse object

Required: No

**EphemeralStorage**

The size of the function's /tmp directory in MB. The default value is 512, but can be any whole number between 512 and 10,240 MB. For more information, see Configuring ephemeral storage (console).

Type: EphemeralStorage object

Required: No

**FileSystemConfigs**

Connection settings for an Amazon EFS file system.

Type: Array of FileSystemConfig objects

Array Members: Maximum number of 1 item.

Required: No

**FunctionArn**

The function's Amazon Resource Name (ARN).

Type: String

Pattern: arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_.]+(:($LATEST|[a-zA-Z0-9-_.]+))?+

Required: No
**FunctionName**

The name of the function.

Type: String


Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-\._]+):((\$LATEST|[a-zA-Z0-9-\._]+))? Required: No

**Handler**

The function that Lambda calls to begin running your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: [\s]+

Required: No

**ImageConfigResponse**

The function's image configuration values.

Type: ImageConfigResponse object

Required: No

**KMSKeyArn**

The AWS KMS key that's used to encrypt the function's environment variables. When Lambda SnapStart is activated, this key is also used to encrypt the function's snapshot. This key is returned only if you've configured a customer managed key.

Type: String

Pattern: (arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-\._]+*:\.*):() Required: No
**LastModified**

The date and time that the function was last updated, in ISO-8601 format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Required: No

**LastUpdateStatus**

The status of the last update that was performed on the function. This is first set to `Successful` after function creation completes.

Type: String

Valid Values: Successful | Failed | InProgress

Required: No

**LastUpdateStatusReason**

The reason for the last update that was performed on the function.

Type: String

Required: No

**LastUpdateStatusReasonCode**

The reason code for the last update that was performed on the function.

Type: String


Required: No
Layers

The function's layers.

Type: Array of Layer objects

Required: No

LoggingConfig

The function's Amazon CloudWatch Logs configuration settings.

Type: LoggingConfig object

Required: No

MasterArn

For Lambda@Edge functions, the ARN of the main function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+--\d{1}: \d{12}:function:[a-zA-Z0-9-_]++(:($LATEST|[a-zA-Z-A-Za-z0-9-]+))?

Required: No

MemorySize

The amount of memory available to the function at runtime.

Type: Integer


Required: No

PackageType

The type of deployment package. Set to Image for container image and set Zip for .zip file archive.

Type: String

Valid Values: Zip | Image

Required: No
**RevisionId**

The latest updated revision of the function or alias.

Type: String

Required: No

**Role**

The function's execution role.

Type: String

Pattern: arn:(aws[a-zA-Z-]*):iam::<d{12}::role/?[a-zA-Z_0-9+=,.@-_/]+

Required: No

**Runtime**

The identifier of the function's runtime. Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17 | ruby3.2 | python3.11 | nodejs20.x | provided.al2023 | python3.12 | java21

Required: No

**RuntimeVersionConfig**

The ARN of the runtime and any errors that occurred.

Type: RuntimeVersionConfig object

Required: No
**SigningJobArn**

The ARN of the signing job.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-]+)([a-z]{2}(-gov)?-[a-z]+-\d{1})?:\d{12}?:(.*)

Required: No

**SigningProfileVersionArn**

The ARN of the signing profile version.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-]+)([a-z]{2}(-gov)?-[a-z]+-\d{1})?:\d{12}?:(.*)

Required: No

**SnapStart**

Set ApplyOn to PublishedVersions to create a snapshot of the initialized execution environment when you publish a function version. For more information, see [Improving startup performance with Lambda SnapStart](#).

Type: SnapStartResponse object

Required: No

**State**

The current state of the function. When the state is Inactive, you can reactivate the function by invoking it.

Type: String

Valid Values: Pending | Active | Inactive | Failed

Required: No

**StateReason**

The reason for the function's current state.
**Type**: String  
**Required**: No

**StateReasonCode**

The reason code for the function's current state. When the code is creating, you can't invoke or modify the function.

**Type**: String

**Valid Values**:  
- Idle  
- Creating  
- Restoring  
- EniLimitExceeded  
- InsufficientRolePermissions  
- InvalidConfiguration  
- InternalError  
- SubnetOutOfIPAddresses  
- InvalidSubnet  
- InvalidSecurityGroup  
- ImageDeleted  
- ImageAccessDenied  
- InvalidImage  
- KMSKeyAccessDenied  
- KMSKeyNotFound  
- InvalidStateKMSKey  
- DisabledKMSKey  
- EFSIOError  
- EFSMountConnectivityError  
- EFSMountFailure  
- EFSMountTimeout  
- InvalidRuntime  
- InvalidZipFileException  
- FunctionError

**Required**: No

**Timeout**

The amount of time in seconds that Lambda allows a function to run before stopping it.

**Type**: Integer

**Valid Range**: Minimum value of 1.

**Required**: No

**TracingConfig**

The function's AWS X-Ray tracing configuration.

**Type**: [TracingConfigResponse](#) object

**Required**: No

**Version**

The version of the Lambda function.

**Type**: String

Pattern: ($LATEST | [0-9]+ )

Required: No

VpcConfig

The function's networking configuration.

Type: VpcConfigResponse object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
FunctionEventInvokeConfig

Contents

DestinationConfig

A destination for events after they have been sent to a function for processing.

Destinations

- **Function** - The Amazon Resource Name (ARN) of a Lambda function.
- **Queue** - The ARN of a standard SQS queue.
- **Topic** - The ARN of a standard SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: DestinationConfig object

Required: No

FunctionArn

The Amazon Resource Name (ARN) of the function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:
\d{12}:function:[a-zA-Z0-9-_.]+(:($LATEST|[a-zA-Z0-9-_.]+))? (no regex)

Required: No

LastModified

The date and time that the configuration was last updated, in Unix time seconds.

Type: Timestamp

Required: No

MaximumEventAgeInSeconds

The maximum age of a request that Lambda sends to a function for processing.

Type: Integer
Valid Range: Minimum value of 60. Maximum value of 21600.

Required: No

**MaximumRetryAttempts**

The maximum number of times to retry when the function returns an error.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 2.

Required: No

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
FunctionUrlConfig

Details about a Lambda function URL.

Contents

AuthType

The type of authentication that your function URL uses. Set to AWS_IAM if you want to restrict access to authenticated users only. Set to NONE if you want to bypass IAM authentication to create a public endpoint. For more information, see Security and auth model for Lambda function URLs.

Type: String

Valid Values: NONE  |  AWS_IAM

Required: Yes

CreationTime

When the function URL was created, in ISO-8601 format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Required: Yes

FunctionArn

The Amazon Resource Name (ARN) of your function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:
\d{12}:function:[a-zA-Z0-9-_.]+(:($LATEST|[a-zA-Z0-9-_.]+))?

Required: Yes

FunctionUrl

The HTTP URL endpoint for your function.

Type: String

Length Constraints: Minimum length of 40. Maximum length of 100.
LastModifiedTime

When the function URL configuration was last updated, in ISO-8601 format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Required: Yes

Cors

The cross-origin resource sharing (CORS) settings for your function URL.

Type: Cors object

Required: No

InvokeMode

Use one of the following options:

- BUFFERED – This is the default option. Lambda invokes your function using the Invoke API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.
- RESPONSE_STREAM – Your function streams payload results as they become available. Lambda invokes your function using the InvokeWithResponseStream API operation. The maximum response payload size is 20 MB, however, you can request a quota increase.

Type: String

Valid Values: BUFFERED | RESPONSE_STREAM

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- **AWS SDK for Java V2**
- **AWS SDK for Ruby V3**
ImageConfig

Configuration values that override the container image Dockerfile settings. For more information, see Container image settings.

Contents

Command

Specifies parameters that you want to pass in with ENTRYPOINT.

Type: Array of strings

Array Members: Maximum number of 1500 items.

Required: No

EntryPoint

Specifies the entry point to their application, which is typically the location of the runtime executable.

Type: Array of strings

Array Members: Maximum number of 1500 items.

Required: No

WorkingDirectory

Specifies the working directory.

Type: String

Length Constraints: Maximum length of 1000.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
ImageConfigError

Error response to GetFunctionConfiguration.

Contents

ErrorCode

Error code.

Type: String

Required: No

Message

Error message.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
ImageConfigResponse

Response to a GetFunctionConfiguration request.

Contents

Error

Error response to GetFunctionConfiguration.

Type: ImageConfigError object

Required: No

ImageConfig

Configuration values that override the container image Dockerfile.

Type: ImageConfig object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
InvokeResponseStreamUpdate

A chunk of the streamed response payload.

Contents

Payload

Data returned by your Lambda function.

Type: Base64-encoded binary data object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
InvokeWithResponseStreamCompleteEvent

A response confirming that the event stream is complete.

Contents

ErrorCode

An error code.

Type: String

Required: No

ErrorDetails

The details of any returned error.

Type: String

Required: No

LogResult

The last 4 KB of the execution log, which is base64-encoded.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
InvokeWithResponseStreamResponseEvent

An object that includes a chunk of the response payload. When the stream has ended, Lambda includes a InvokeComplete object.

Contents

InvokeComplete

An object that's returned when the stream has ended and all the payload chunks have been returned.

Type: InvokeWithResponseStreamCompleteEvent object

Required: No

PayloadChunk

A chunk of the streamed response payload.

Type: InvokeResponseStreamUpdate object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
Layer

An AWS Lambda layer.

Contents

Arn

The Amazon Resource Name (ARN) of the function layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:$d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+

Required: No

CodeSize

The size of the layer archive in bytes.

Type: Long

Required: No

SigningJobArn

The Amazon Resource Name (ARN) of a signing job.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-]+):([a-zA-Z]{2}(-gov)?-[a-zA-Z]+-\d{1})?:\d{12}?:(.*)

Required: No

SigningProfileVersionArn

The Amazon Resource Name (ARN) for a signing profile version.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-]+):([a-zA-Z]{2}(-gov)?-[a-zA-Z]+-\d{1})?:\d{12}?:(.*)
Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
LayersListItem

Details about an AWS Lambda layer.

Contents

LatestMatchingVersion

The newest version of the layer.

Type: LayerVersionsListItem object

Required: No

LayerArn

The Amazon Resource Name (ARN) of the function layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-\_]+|

Required: No

LayerName

The name of the layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-\_]+)+|[a-zA-Z0-9-\_]+|

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
**LayerVersionContentInput**

A ZIP archive that contains the contents of an **AWS Lambda layer**. You can specify either an Amazon S3 location, or upload a layer archive directly.

**Contents**

**S3Bucket**

The Amazon S3 bucket of the layer archive.

Type: String


Pattern: ^[0-9A-Za-z\.\-_]*(?<!\.)$

Required: No

**S3Key**

The Amazon S3 key of the layer archive.

Type: String


Required: No

**S3ObjectVersion**

For versioned objects, the version of the layer archive object to use.

Type: String


Required: No

**ZipFile**

The base64-encoded contents of the layer archive. AWS SDK and AWS CLI clients handle the encoding for you.

Type: Base64-encoded binary data object
Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
LayerVersionContentOutput

Details about a version of an AWS Lambda layer.

Contents

CodeSha256

The SHA-256 hash of the layer archive.

Type: String
Required: No

CodeSize

The size of the layer archive in bytes.

Type: Long
Required: No

Location

A link to the layer archive in Amazon S3 that is valid for 10 minutes.

Type: String
Required: No

SigningJobArn

The Amazon Resource Name (ARN) of a signing job.

Type: String
Required: No

SigningProfileVersionArn

The Amazon Resource Name (ARN) for a signing profile version.

Type: String
Required: No
See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
LayerVersionsListItem

Details about a version of an AWS Lambda layer.

Contents

CompatibleArchitectures

A list of compatible instruction set architectures.

Type: Array of strings

Array Members: Maximum number of 2 items.

Valid Values: x86_64 | arm64

Required: No

CompatibleRuntimes

The layer's compatible runtimes.

The following list includes deprecated runtimes. For more information, see Runtime deprecation policy.

Type: Array of strings

Array Members: Maximum number of 15 items.

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17 | ruby3.2 | python3.11 | nodejs20.x | provided.al2023 | python3.12 | java21

Required: No

CreatedDate

The date that the version was created, in ISO 8601 format. For example, 2018-11-27T15:10:45.123+0000.
Type: String

Required: No

**Description**

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

**LayerVersionArn**

The ARN of the layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `arn:([a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+):\d{12}:layer:[a-zA-Z0-9-_.]+:[0-9]+`

Required: No

**LicenseInfo**

The layer's open-source license.

Type: String

Length Constraints: Maximum length of 512.

Required: No

**Version**

The version number.

Type: Long

Required: No
See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
LoggingConfig

The function's Amazon CloudWatch Logs configuration settings.

Contents

ApplicationLogLevel

Set this property to filter the application logs for your function that Lambda sends to CloudWatch. Lambda only sends application logs at the selected level and lower.

Type: String

Valid Values: TRACE | DEBUG | INFO | WARN | ERROR | FATAL

Required: No

LogFormat

The format in which Lambda sends your function’s application and system logs to CloudWatch. Select between plain text and structured JSON.

Type: String

Valid Values: JSON | Text

Required: No

LogGroup

The name of the Amazon CloudWatch log group the function sends logs to. By default, Lambda functions send logs to a default log group named /aws/lambda/<function name>. To use a different log group, enter an existing log group or enter a new log group name.

Type: String


Pattern: [\.-_/#A-Za-z0-9]+

Required: No

SystemLogLevel

Set this property to filter the system logs for your function that Lambda sends to CloudWatch. Lambda only sends system logs at the selected level and lower.
Type: String

Valid Values: DEBUG | INFO | WARN

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
OnFailure

A destination for events that failed processing.

Contents

Destination

The Amazon Resource Name (ARN) of the destination resource.

To retain records of asynchronous invocations, you can configure an Amazon SNS topic, Amazon SQS queue, Lambda function, or Amazon EventBridge event bus as the destination.

To retain records of failed invocations from Kinesis and DynamoDB event sources, you can configure an Amazon SNS topic or Amazon SQS queue as the destination.

To retain records of failed invocations from self-managed Kafka or Amazon MSK, you can configure an Amazon SNS topic or Amazon SQS queue as the destination.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 350.

Pattern: ^$|arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9-]*)+([a-zA-Z]{2}(-gov)?-[a-zA-Z]+-\d{12}):([\d{12}])?:(.*)

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
OnSuccess

A destination for events that were processed successfully.

Contents

Destination

The Amazon Resource Name (ARN) of the destination resource.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 350.

Pattern: ^$|arn:(aws[a-zA-Z0-9-]*):[(a-zA-Z0-9\-\:])+:([a-z]{2}(-gov)?-[a-zA-Z]+-\d{1})?(:\d{12})?(:(.*)

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
ProvisionedConcurrencyConfigListItem

Details about the provisioned concurrency configuration for a function alias or version.

Contents

AllocatedProvisionedConcurrentExecutions

The amount of provisioned concurrency allocated. When a weighted alias is used during linear and canary deployments, this value fluctuates depending on the amount of concurrency that is provisioned for the function versions.

Type: Integer

Valid Range: Minimum value of 0.

Required: No

AvailableProvisionedConcurrentExecutions

The amount of provisioned concurrency available.

Type: Integer

Valid Range: Minimum value of 0.

Required: No

FunctionArn

The Amazon Resource Name (ARN) of the alias or version.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-\_]+(:($\{\$LATEST|[a-zA-Z0-9-\_]+))?

Required: No

LastModified

The date and time that a user last updated the configuration, in ISO 8601 format.

Type: String
Required: No

**RequestedProvisionedConcurrentExecutions**

The amount of provisioned concurrency requested.

Type: Integer

Valid Range: Minimum value of 1.

Required: No

**Status**

The status of the allocation process.

Type: String

Valid Values: IN_PROGRESS | READY | FAILED

Required: No

**StatusReason**

For failed allocations, the reason that provisioned concurrency could not be allocated.

Type: String

Required: No

**See Also**

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
RuntimeVersionConfig

The ARN of the runtime and any errors that occurred.

Contents

Error

Error response when Lambda is unable to retrieve the runtime version for a function.

Type: RuntimeVersionError object

Required: No

RuntimeVersionArn

The ARN of the runtime version you want the function to use.

Type: String


Pattern: ^arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}((-gov)|(-iso(b?)?)?-[a-z]+-\d{1}::runtime:.+${

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
RuntimeVersionError

Any error returned when the runtime version information for the function could not be retrieved.

Contents

ErrorCode

The error code.

Type: String

Required: No

Message

The error message.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
ScalingConfig

(Amazon SQS only) The scaling configuration for the event source. To remove the configuration, pass an empty value.

Contents

MaximumConcurrency

Limits the number of concurrent instances that the Amazon SQS event source can invoke.

Type: Integer


Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
SelfManagedEventSource

The self-managed Apache Kafka cluster for your event source.

Contents

Endpoints

The list of bootstrap servers for your Kafka brokers in the following format:
"KAFKA_BOOTSTRAP_SERVERS": ["abc.xyz.com:xxxx","abc2.xyz.com:xxxx"].

Type: String to array of strings map

Map Entries: Maximum number of 2 items.

Valid Keys: KAFKA_BOOTSTRAP_SERVERS

Array Members: Minimum number of 1 item. Maximum number of 10 items.

Length Constraints: Minimum length of 1. Maximum length of 300.

Pattern: ^((\[a-zA-Z0-9]([a-zA-Z0-9]\-[a-zA-Z0-9]*[a-zA-Z0-9])\.|(\[A-Za-z0-9]\-[A-Za-z0-9]\-[A-Za-z0-9]*[A-Za-z0-9])):([0-9]{1,5})

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
SelfManagedKafkaEventSourceConfig

Specific configuration settings for a self-managed Apache Kafka event source.

Contents

ConsumerGroupId

The identifier for the Kafka consumer group to join. The consumer group ID must be unique among all your Kafka event sources. After creating a Kafka event source mapping with the consumer group ID specified, you cannot update this value. For more information, see Customizable consumer group ID.

Type: String


Pattern: [a-zA-Z0-9-\/*:_+=.@-]*

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
SnapStart

The function's Lambda SnapStart setting. Set ApplyOn to PublishedVersions to create a snapshot of the initialized execution environment when you publish a function version.

Contents

ApplyOn

Set to PublishedVersions to create a snapshot of the initialized execution environment when you publish a function version.

Type: String

Valid Values: PublishedVersions | None

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)
SnapStartResponse

The function's SnapStart setting.

Contents

ApplyOn

When set to PublishedVersions, Lambda creates a snapshot of the execution environment when you publish a function version.

Type: String

Valid Values: PublishedVersions | None

Required: No

OptimizationStatus

When you provide a qualified Amazon Resource Name (ARN), this response element indicates whether SnapStart is activated for the specified function version.

Type: String

Valid Values: On | Off

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
SourceAccessConfiguration

To secure and define access to your event source, you can specify the authentication protocol, VPC components, or virtual host.

Contents

Type

The type of authentication protocol, VPC components, or virtual host for your event source. For example: "Type": "SASL_SCRAM_512_AUTH".

- **BASIC_AUTH** – (Amazon MQ) The AWS Secrets Manager secret that stores your broker credentials.
- **BASIC_AUTH** – (Self-managed Apache Kafka) The Secrets Manager ARN of your secret key used for SASL/PLAIN authentication of your Apache Kafka brokers.
- **VPC_SUBNET** – (Self-managed Apache Kafka) The subnets associated with your VPC. Lambda connects to these subnets to fetch data from your self-managed Apache Kafka cluster.
- **VPC_SECURITY_GROUP** – (Self-managed Apache Kafka) The VPC security group used to manage access to your self-managed Apache Kafka brokers.
- **SASL_SCRAM_256_AUTH** – (Self-managed Apache Kafka) The Secrets Manager ARN of your secret key used for SASL SCRAM-256 authentication of your self-managed Apache Kafka brokers.
- **SASL_SCRAM_512_AUTH** – (Amazon MSK, Self-managed Apache Kafka) The Secrets Manager ARN of your secret key used for SASL SCRAM-512 authentication of your self-managed Apache Kafka brokers.
- **VIRTUAL_HOST** – (RabbitMQ) The name of the virtual host in your RabbitMQ broker. Lambda uses this RabbitMQ host as the event source. This property cannot be specified in an UpdateEventSourceMapping API call.
- **CLIENT_CERTIFICATE_TLS_AUTH** – (Amazon MSK, Self-managed Apache Kafka) The Secrets Manager ARN of your secret key containing the certificate chain (X.509 PEM), private key (PKCS#8 PEM), and private key password (optional) used for mutual TLS authentication of your MSK/Apache Kafka brokers.
- **SERVER_ROOT_CA_CERTIFICATE** – (Self-managed Apache Kafka) The Secrets Manager ARN of your secret key containing the root CA certificate (X.509 PEM) used for TLS encryption of your Apache Kafka brokers.
Type: String

Valid Values: BASIC_AUTH | VPC_SUBNET | VPC_SECURITY_GROUP | SASL_SCRAM_512_AUTH | SASL_SCRAM_256_AUTH | VIRTUAL_HOST | CLIENT_CERTIFICATE_TLS_AUTH | SERVER_ROOT_CA_CERTIFICATE

Required: No

URI

The value for your chosen configuration in Type. For example: "URI": "arn:aws:secretsmanager:us-east-1:01234567890:secret:MyBrokerSecretName".

Type: String


Pattern: [a-zA-Z0-9-/\:*_:+=.@-]*

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
TracingConfig

The function's AWS X-Ray tracing configuration. To sample and record incoming requests, set Mode to Active.

Contents

Mode

The tracing mode.

Type: String

Valid Values: Active | PassThrough

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
TracingConfigResponse

The function's AWS X-Ray tracing configuration.

Contents

Mode

The tracing mode.

Type: String

Valid Values: Active | PassThrough

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java V2
- AWS SDK for Ruby V3
The VPC security groups and subnets that are attached to a Lambda function. For more information, see [Configuring a Lambda function to access resources in a VPC](#).

### Contents

**Ipv6AllowedForDualStack**

Allows outbound IPv6 traffic on VPC functions that are connected to dual-stack subnets.

Type: Boolean

Required: No

**SecurityGroupIds**

A list of VPC security group IDs.

Type: Array of strings

Array Members: Maximum number of 5 items.

Required: No

**SubnetIds**

A list of VPC subnet IDs.

Type: Array of strings

Array Members: Maximum number of 16 items.

Required: No

### See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
• **AWS SDK for Ruby V3**
VpcConfigResponse

The VPC security groups and subnets that are attached to a Lambda function.

Contents

Ipv6AllowedForDualStack

Allows outbound IPv6 traffic on VPC functions that are connected to dual-stack subnets.

Type: Boolean

Required: No

SecurityGroupIds

A list of VPC security group IDs.

Type: Array of strings

Array Members: Maximum number of 5 items.

Required: No

SubnetIds

A list of VPC subnet IDs.

Type: Array of strings

Array Members: Maximum number of 16 items.

Required: No

VpcId

The ID of the VPC.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:
Certificate errors when using an SDK

Because AWS SDKs use the CA certificates from your computer, changes to the certificates on the AWS servers can cause connection failures when you attempt to use an SDK. You can prevent these failures by keeping your computer's CA certificates and operating system up-to-date. If you encounter this issue in a corporate environment and do not manage your own computer, you might need to ask an administrator to assist with the update process. The following list shows minimum operating system and Java versions:

- Microsoft Windows versions that have updates from January 2005 or later installed contain at least one of the required CAs in their trust list.
- Mac OS X 10.4 with Java for Mac OS X 10.4 Release 5 (February 2007), Mac OS X 10.5 (October 2007), and later versions contain at least one of the required CAs in their trust list.
- Red Hat Enterprise Linux 5 (March 2007), 6, and 7 and CentOS 5, 6, and 7 all contain at least one of the required CAs in their default trusted CA list.
- Java 1.4.2_12 (May 2006), 5 Update 2 (March 2005), and all later versions, including Java 6 (December 2006), 7, and 8, contain at least one of the required CAs in their default trusted CA list.

When accessing the AWS Lambda management console or AWS Lambda API endpoints, whether through browsers or programmatically, you will need to ensure your client machines support any of the following CAs:

- Amazon Root CA 1
- Starfield Services Root Certificate Authority - G2
- Starfield Class 2 Certification Authority

Root certificates from the first two authorities are available from Amazon trust services, but keeping your computer up-to-date is the more straightforward solution. To learn more about ACM-provided certificates, see AWS Certificate Manager FAQs.
Common Errors

This section lists the errors common to the API actions of all AWS services. For errors specific to an API action for this service, see the topic for that API action.

AccessDeniedException

You do not have sufficient access to perform this action.

HTTP Status Code: 400

IncompleteSignature

The request signature does not conform to AWS standards.

HTTP Status Code: 400

InternalFailure

The request processing has failed because of an unknown error, exception or failure.

HTTP Status Code: 500

InvalidAction

The action or operation requested is invalid. Verify that the action is typed correctly.

HTTP Status Code: 400

InvalidClientTokenId

The X.509 certificate or AWS access key ID provided does not exist in our records.

HTTP Status Code: 403

Unauthorized

You do not have permission to perform this action.

HTTP Status Code: 400

OptInRequired

The AWS access key ID needs a subscription for the service.

HTTP Status Code: 403
RequestExpired

The request reached the service more than 15 minutes after the date stamp on the request or more than 15 minutes after the request expiration date (such as for pre-signed URLs), or the date stamp on the request is more than 15 minutes in the future.

HTTP Status Code: 400

ServiceUnavailable

The request has failed due to a temporary failure of the server.

HTTP Status Code: 503

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationError

The input fails to satisfy the constraints specified by an AWS service.

HTTP Status Code: 400

Common Parameters

The following list contains the parameters that all actions use for signing Signature Version 4 requests with a query string. Any action-specific parameters are listed in the topic for that action. For more information about Signature Version 4, see Signing AWS API requests in the IAM User Guide.

Action

The action to be performed.

Type: string

Required: Yes

Version

The API version that the request is written for, expressed in the format YYYY-MM-DD.
**X-Amz-Algorithm**

The hash algorithm that you used to create the request signature.

Condition: Specify this parameter when you include authentication information in a query string instead of in the HTTP authorization header.

Type: string

Valid Values: AWS4-HMAC-SHA256

Required: Conditional

**X-Amz-Credential**

The credential scope value, which is a string that includes your access key, the date, the region you are targeting, the service you are requesting, and a termination string ("aws4_request"). The value is expressed in the following format: access_key/YYYYMMDD/region/service/aws4_request.

For more information, see [Create a signed AWS API request](https://docs.aws.amazon.com/IAM/latest/UserGuidecréate-a-signed-aws-api-request.html) in the *IAM User Guide*.

Condition: Specify this parameter when you include authentication information in a query string instead of in the HTTP authorization header.

Type: string

Required: Conditional

**X-Amz-Date**

The date that is used to create the signature. The format must be ISO 8601 basic format (YYYYMMDD'T'HHMMSS'Z'). For example, the following date time is a valid X-Amz-Date value: 20120325T120000Z.

Condition: X-Amz-Date is optional for all requests; it can be used to override the date used for signing requests. If the Date header is specified in the ISO 8601 basic format, X-Amz-Date is not required. When X-Amz-Date is used, it always overrides the value of the Date header. For more information, see [Elements of an AWS API request signature](https://docs.aws.amazon.com/IAM/latest/UserGuideelements-of-an-aws-api-request-signature.html) in the *IAM User Guide*.
Type: string

Required: Conditional

**X-Amz-Security-Token**

The temporary security token that was obtained through a call to AWS Security Token Service (AWS STS). For a list of services that support temporary security credentials from AWS STS, see [AWS services that work with IAM](https://docs.aws.amazon.com/iam/latest/userguide/what-is-users.html) in the *IAM User Guide*.

Condition: If you're using temporary security credentials from AWS STS, you must include the security token.

Type: string

Required: Conditional

**X-Amz-Signature**

Specifies the hex-encoded signature that was calculated from the string to sign and the derived signing key.

Condition: Specify this parameter when you include authentication information in a query string instead of in the HTTP authorization header.

Type: string

Required: Conditional

**X-Amz-SignedHeaders**

Specifies all the HTTP headers that were included as part of the canonical request. For more information about specifying signed headers, see [Create a signed AWS API request](https://docs.aws.amazon.com/iam/latest/userguide/create-signed-requests.html) in the *IAM User Guide*.

Condition: Specify this parameter when you include authentication information in a query string instead of in the HTTP authorization header.

Type: string

Required: Conditional
## Document history

The following table describes the important changes to the *AWS Lambda Developer Guide* since May 2018. For notification about updates to this documentation, subscribe to the [RSS feed](#).

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Python 3.12 runtime</strong></td>
<td>Lambda now supports Python 3.12 as a managed runtime and container base image. For more information, see <a href="#">Python 3.12 runtime now available in AWS Lambda</a> on the AWS Compute Blog.</td>
<td>December 14, 2023</td>
</tr>
<tr>
<td><strong>Java 21 runtime</strong></td>
<td>Lambda now supports Java 21 as a managed runtime and container base image (java21).</td>
<td>November 16, 2023</td>
</tr>
<tr>
<td><strong>Node.js 20.x runtime</strong></td>
<td>Lambda now supports Node.js 20 as a managed runtime and container base image (nodejs20.x). For more information, see <a href="#">Node.js 20.x runtime now available in AWS Lambda</a> on the AWS Compute Blog.</td>
<td>November 14, 2023</td>
</tr>
<tr>
<td><strong>provided.al2023 runtime</strong></td>
<td>Lambda now supports Amazon Linux 2023 as a managed runtime and container base image. For more information, see <a href="#">Introducing the Amazon Linux 2023 runtime for AWS</a></td>
<td>November 9, 2023</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>IPv6 support for dual-stack subnets</td>
<td>Lambda now supports outbound IPv6 traffic to dual-stack subnets. For more information, see <a href="#">IPv6 support</a>.</td>
<td>October 12, 2023</td>
</tr>
<tr>
<td>Testing serverless functions and applications</td>
<td>Learn about techniques to debug and automate testing serverless functions in the cloud. There is now a testing chapter and resources included in the Python and Typescript language sections. For details, see <a href="#">Testing serverless functions and applications</a>.</td>
<td>June 16, 2023</td>
</tr>
<tr>
<td>Ruby 3.2 runtime</td>
<td>Lambda now supports a new runtime for Ruby 3.2. For more information, see <a href="#">Building Lambda functions with Ruby</a>.</td>
<td>June 7, 2023</td>
</tr>
<tr>
<td>Response streaming</td>
<td>Lambda now supports streaming responses from functions. For more information, see <a href="#">Configuring a Lambda function to stream responses</a>.</td>
<td>April 6, 2023</td>
</tr>
<tr>
<td>Asynchronous invocation metrics</td>
<td>Lambda releases asynchronous invocation metrics. For more information, see <a href="#">Asynchronous invocation metrics</a>.</td>
<td>February 9, 2023</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td><strong>Runtime version controls</strong></td>
<td>Lambda releases new runtime versions that include security updates, bug fixes, and new features. You can now control when your functions get updated to the new runtime versions. For more information, see <a href="#">Lambda runtime updates</a>.</td>
<td>January 23, 2023</td>
</tr>
<tr>
<td><strong>Lambda SnapStart</strong></td>
<td>Use Lambda SnapStart to reduce startup time for Java functions without provisioning additional resources or implementing complex performance optimizations. For more information, see <a href="#">Improving startup performance with Lambda SnapStart</a>.</td>
<td>November 28, 2022</td>
</tr>
<tr>
<td><strong>Node.js 18 runtime</strong></td>
<td>Lambda now supports a new runtime for Node.js 18. Node.js 18 uses Amazon Linux 2. For details, see <a href="#">Building Lambda functions with Node.js</a>.</td>
<td>November 18, 2022</td>
</tr>
<tr>
<td><strong>lambda:SourceFunctionArn</strong></td>
<td>For an AWS resource, the <code>lambda:SourceFunctionArn</code> condition key filters access to the resource by the ARN of a Lambda function. For details, see <a href="#">Working with Lambda execution environment credentials</a>.</td>
<td>July 1, 2022</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td><strong>Node.js 16 runtime</strong></td>
<td>Lambda now supports a new runtime for Node.js 16. Node.js 16 uses Amazon Linux 2. For details, see <a href="#">Building Lambda functions with Node.js</a>.</td>
<td>May 11, 2022</td>
</tr>
<tr>
<td><strong>Lambda function URLs</strong></td>
<td>Lambda now supports function URLs, which are dedicated HTTP(S) endpoints for Lambda functions. For details, see <a href="#">Lambda function URLs</a>.</td>
<td>April 6, 2022</td>
</tr>
<tr>
<td><strong>Shared test events in the AWS Lambda console</strong></td>
<td>Lambda now supports sharing test events with other users in the same AWS account. For details, see <a href="#">Testing Lambda functions in the console</a>.</td>
<td>March 16, 2022</td>
</tr>
<tr>
<td><strong>PrincipalOrgId in resource-based policies</strong></td>
<td>Lambda now supports granting permissions to an organization in AWS Organizations. For details, see <a href="#">Using resource-based policies for AWS Lambda</a>.</td>
<td>March 11, 2022</td>
</tr>
<tr>
<td><strong>.NET 6 runtime</strong></td>
<td>Lambda now supports a new runtime for .NET 6. For details, see <a href="#">Lambda runtimes</a>.</td>
<td>February 23, 2022</td>
</tr>
<tr>
<td><strong>Event filtering for Kinesis, DynamoDB, and Amazon SQS event sources</strong></td>
<td>Lambda now supports event filtering for Kinesis, DynamoDB, and Amazon SQS event sources. For details, see <a href="#">Lambda event filtering</a>.</td>
<td>November 24, 2021</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td><strong>mTLS authentication for Amazon MSK and self-managed Apache Kafka event sources</strong></td>
<td>Lambda now supports mTLS authentication for Amazon MSK and self-managed Apache Kafka event sources. For details, see <a href="#">Using Lambda with Amazon MSK</a>.</td>
<td>November 19, 2021</td>
</tr>
<tr>
<td><strong>Lambda on Graviton2</strong></td>
<td>Lambda now supports Graviton2 for functions using arm64 architecture. For details, see <a href="#">Lambda instruction set architectures</a>.</td>
<td>September 29, 2021</td>
</tr>
<tr>
<td><strong>Python 3.9 runtime</strong></td>
<td>Lambda now supports a new runtime for Python 3.9. For details, see <a href="#">Lambda runtimes</a>.</td>
<td>August 16, 2021</td>
</tr>
<tr>
<td><strong>New runtime versions for Node.js, Python, and Java</strong></td>
<td>New runtime versions are available for Node.js, Python, and Java. For details, see <a href="#">Lambda runtimes</a>.</td>
<td>July 21, 2021</td>
</tr>
<tr>
<td><strong>Support for RabbitMQ as an event source on Lambda</strong></td>
<td>Lambda now supports Amazon MQ for RabbitMQ as an event source. Amazon MQ is a managed message broker service for Apache ActiveMQ and RabbitMQ that makes it easy to set up and operate message brokers in the cloud. For details, see <a href="#">Using Lambda with Amazon MQ</a>.</td>
<td>July 7, 2021</td>
</tr>
</tbody>
</table>
### SASL/PLAIN authentication for self-managed Kafka on Lambda

SASL/PLAIN is now a supported authentication mechanism for self-managed Kafka event sources on Lambda. Customers already using SASL/PLAIN on their self-managed Kafka cluster can now easily use Lambda to build consumer applications without having to modify the way they authenticate. For details, see [Using Lambda with self-managed Apache Kafka](#).

**June 29, 2021**

### Lambda Extensions API

General availability for Lambda extensions. Use extensions to augment your Lambda functions. You can use extensions provided by Lambda Partners, or you can create your own Lambda extensions. For details, see [Lambda Extensions API](#).

**May 24, 2021**

### New Lambda console experience

The Lambda console has been redesigned to improve performance and consistency.

**March 2, 2021**

### Node.js 14 runtime

Lambda now supports a new runtime for Node.js 14. Node.js 14 uses Amazon Linux 2. For details, see [Building Lambda functions with Node.js](#).

**January 27, 2021**
<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lambda container images</strong></td>
<td>Lambda now supports functions defined as container images. You can combine the flexibility of container tooling with the agility and operational simplicity of Lambda to build applications. For details, see <a href="#">Using container images with Lambda</a>.</td>
<td>December 1, 2020</td>
</tr>
<tr>
<td><strong>Code signing for Lambda functions</strong></td>
<td>Lambda now supports code signing. Administrators can configure Lambda functions to accept only signed code on deployment. Lambda checks the signatures to ensure that the code is not altered or tampered. Additionally, Lambda ensures that the code is signed by trusted developers before accepting the deployment. For details, see <a href="#">Configuring code signing for Lambda</a>.</td>
<td>November 23, 2020</td>
</tr>
<tr>
<td><strong>Preview: Lambda Runtime Logs API</strong></td>
<td>Lambda now supports the Runtime Logs API. Lambda extensions can use the Logs API to subscribe to log streams in the execution environment. For details, see <a href="#">Lambda Runtime Logs API</a>.</td>
<td>November 12, 2020</td>
</tr>
</tbody>
</table>
New event source to for Amazon MQ
Lambda now supports Amazon MQ as an event source. Use a Lambda function to process records from your Amazon MQ message broker. For details, see Using Lambda with Amazon MQ.
November 5, 2020

Preview: Lambda Extensions API
Use Lambda extensions to augment your Lambda functions. You can use extensions provided by Lambda Partners, or you can create your own Lambda extensions. For details, see Lambda Extensions API.
October 8, 2020

Support for Java 8 and custom runtimes on AL2
Lambda now supports Java 8 and custom runtimes on Amazon Linux 2. For details, see Lambda runtimes.
August 12, 2020

New event source for Amazon Managed Streaming for Apache Kafka
Lambda now supports Amazon MSK as an event source. Use a Lambda function with Amazon MSK to process records in a Kafka topic. For details, see Using Lambda with Amazon MSK.
August 11, 2020
<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAM condition keys for Amazon VPC settings</td>
<td>You can now use Lambda-specific condition keys for VPC settings. For example, you can require that all functions in your organization are connected to a VPC. You can also specify the subnets and security groups that the function's users can and can't use. For details, see Configuring VPC for IAM functions.</td>
<td>August 10, 2020</td>
</tr>
<tr>
<td>Concurrency settings for Kinesis HTTP/2 stream consumers</td>
<td>You can now use the following concurrency settings for Kinesis consumers with enhanced fan-out (HTTP/2 streams): ParallelizationFactor, MaximumRetryAttempts, MaximumRecordAgeInSeconds, DestinationConfig, and BisectBatchOnFunctionError. For details, see Using AWS Lambda with Amazon Kinesis.</td>
<td>July 7, 2020</td>
</tr>
<tr>
<td>Batch window for Kinesis HTTP/2 stream consumers</td>
<td>You can now configure a batch window (MaximumBatchingWindowInSeconds) for HTTP/2 streams. Lambda reads records from the stream until it has gathered a full batch, or until the batch window expires. For details, see Using AWS Lambda with Amazon Kinesis.</td>
<td>June 18, 2020</td>
</tr>
</tbody>
</table>
Support for Amazon EFS file systems

You can now connect an Amazon EFS file system to your Lambda functions for shared network file access. For details, see Configuring file system access for Lambda functions.

June 16, 2020

AWS CDK sample applications in the Lambda console

The Lambda console now includes sample applications that use the AWS Cloud Development Kit (AWS CDK) for TypeScript. The AWS CDK is a framework that enables you to define your application resources in TypeScript, Python, Java, or .NET. For a tutorial on creating applications, see Creating an application with continuous delivery in the Lambda console.

June 1, 2020

Support for .NET Core 3.1.0 runtime in AWS Lambda

AWS Lambda now supports the .NET Core 3.1.0 runtime. For details, see .NET Core CLI.

March 31, 2020

Support for API Gateway HTTP APIs

Updated and expanded documentation for using Lambda with API Gateway, including support for HTTP APIs. Added a sample application that creates an API and function with AWS CloudFormation. For details, see Using Lambda with Amazon API Gateway.

March 23, 2020
| Ruby 2.7                                                                 | A new runtime is available for Ruby 2.7, ruby2.7, which is the first Ruby runtime to use Amazon Linux 2. For details, see Building Lambda functions with Ruby. | February 19, 2020 |
| Concurrency metrics                                                      | Lambda now reports the ConcurrentExecutions metric for all functions, aliases, and versions. You can view a graph for this metric on the monitoring page for your function. Previously, ConcurrentExecutions was only reported at the account level and for functions that use reserved concurrency. For details, see AWS Lambda function metrics. | February 18, 2020 |
Update to function states

Function states are now enforced for all functions by default. When you connect a function to a VPC, Lambda creates shared elastic network interfaces. This enables your function to scale up without creating additional network interfaces. During this time, you can't perform additional operations on the function, including updating its configuration and publishing versions. In some cases, invocation is also impacted. Details about a function's current state are available from the Lambda API.

This update is being released in phases. For details, see Updated Lambda states lifecycle for VPC networking on the AWS Compute Blog. For more information about states, see AWS Lambda function states.
Updates to function configuration API output

Added reason codes to StateReasonCode (InvalidSubnet, InvalidSecurityGroup) and LastUpdateStatusReasonCode (SubnetOutOfIPAddresses, InvalidSubnet, InvalidSecurityGroup) for functions that connect to a VPC. For more information about states, see AWS Lambda function states.

Provisioned concurrency

You can now allocate provisioned concurrency for a function version or alias. Provisioned concurrency enables a function to scale without fluctuations in latency. For details, see Managing concurrency for a Lambda function.

Create a database proxy

You can now use the Lambda console to create a database proxy for a Lambda function. A database proxy enables a function to reach high concurrency levels without exhausting database connections. For details, see Configuring database access for a Lambda function.

Percentiles support for the duration metric

You can now filter the duration metric based on percentiles. For details, see AWS Lambda metrics.

January 20, 2020

December 3, 2019

December 3, 2019

November 26, 2019
Increased concurrency for stream event sources

A new option for DynamoDB stream and Kinesis stream event source mappings enables you to process more than one batch at a time from each shard. When you increase the number of concurrent batches per shard, your function's concurrency can be up to 10 times the number of shards in your stream. For details, see Lambda event source mapping.

Function states

When you create or update a function, it enters a pending state while Lambda provisions resources to support it. If you connect your function to a VPC, Lambda can create a shared elastic network interface right away, instead of creating network interfaces when your function is invoked. This results in better performance for VPC-connected functions, but might require an update to your automation. For details, see AWS Lambda function states.
<p>| Error handling options for asynchronous invocation | New configuration options are available for asynchronous invocation. You can configure Lambda to limit retries and set a maximum event age. For details, see <a href="#">Configuring error handling for asynchronous invocation</a>. | November 25, 2019 |
| Error handling for stream event sources | New configuration options are available for event source mappings that read from streams. You can configure <a href="#">DynamoDB stream</a> and <a href="#">Kinesis stream</a> event source mappings to limit retries and set a maximum record age. When errors occur, you can configure the event source mapping to split batches before retrying, and to send invocation records for failed batches to a queue or topic. For details, see <a href="#">Lambda event source mapping</a>. | November 25, 2019 |</p>
<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Destinations for asynchronous invocation</strong></td>
<td>You can now configure Lambda to send records of asynchronous invocations to another service. Invocation records contain details about the event, context, and function response. You can send invocation records to an SQS queue, SNS topic, Lambda function, or EventBridge event bus. For details, see Configuring destinations for asynchronous invocation.</td>
<td>November 25, 2019</td>
</tr>
<tr>
<td><strong>New runtimes for Node.js, Python, and Java</strong></td>
<td>New runtimes are available for Node.js 12, Python 3.8, and Java 11. For details, see Lambda runtimes.</td>
<td>November 18, 2019</td>
</tr>
<tr>
<td><strong>FIFO queue support for Amazon SQS event sources</strong></td>
<td>You can now create an event source mapping that reads from a first-in, first-out (FIFO) queue. Previously, only standard queues were supported. For details, see Using Lambda with Amazon SQS.</td>
<td>November 18, 2019</td>
</tr>
<tr>
<td><strong>Create applications in the Lambda console</strong></td>
<td>Application creation in the Lambda console is now generally available. For instructions, see Creating an application with continuous delivery in the Lambda console.</td>
<td>October 31, 2019</td>
</tr>
</tbody>
</table>
Create applications in the Lambda console (beta)  

You can now create a Lambda application with an integrated continuous delivery pipeline in the Lambda console. The console provides sample applications that you can use as a starting point for your own project. Choose between AWS CodeCommit and GitHub for source control. Each time you push changes to your repository, the included pipeline builds and deploys them automatically. For instructions, see Creating an application with continuous delivery in the Lambda console.
Performance improvements for VPC-connected functions

Lambda now uses a new type of elastic network interface that is shared by all functions in a virtual private cloud (VPC) subnet. When you connect a function to a VPC, Lambda creates a network interface for each combination of security group and subnet that you choose. When the shared network interfaces are available, the function no longer needs to create additional network interfaces as it scales up. This dramatically improves startup times. For details, see Configuring a Lambda function to access resources in a VPC.

Stream batch settings

You can now configure a batch window for Amazon DynamoDB and Amazon Kinesis event source mappings. Configure a batch window of up to five minutes to buffer incoming records until a full batch is available. This reduces the number of times that your function is invoked when the stream is less active.

September 3, 2019

August 29, 2019
<table>
<thead>
<tr>
<th><strong>CloudWatch Logs Insights integration</strong></th>
<th>The monitoring page in the Lambda console now includes reports from Amazon CloudWatch Logs Insights. For details, see <a href="#">Monitoring functions in the AWS Lambda console</a>.</th>
<th>June 18, 2019</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Amazon Linux 2018.03</strong></td>
<td>The Lambda execution environment is being updated to use Amazon Linux 2018.03. For details, see <a href="#">Execution environment</a>.</td>
<td>May 21, 2019</td>
</tr>
<tr>
<td><strong>Node.js 10</strong></td>
<td>A new runtime is available for Node.js 10, nodejs10.x. This runtime uses Node.js 10.15 and will be updated with the latest point release of Node.js 10 periodically. Node.js 10 is also the first runtime to use Amazon Linux 2. For details, see <a href="#">Building Lambda functions with Node.js</a>.</td>
<td>May 13, 2019</td>
</tr>
<tr>
<td><strong>GetLayerVersionByArn API</strong></td>
<td>Use the <a href="#">GetLayerVersionByArn</a> API to download layer version information with the version ARN as input. Compared to GetLayerVersion, GetLayerVersionByArn lets you use the ARN directly instead of parsing it to get the layer name and version number.</td>
<td>April 25, 2019</td>
</tr>
</tbody>
</table>
Ruby

AWS Lambda now supports Ruby 2.5 with a new runtime. For details, see Building Lambda functions with Ruby.

Layers

With Lambda layers, you can package and deploy libraries, custom runtimes, and other dependencies separately from your function code. Share your layers with your other accounts or the whole world. For details, see Lambda layers.

Custom runtimes

Build a custom runtime to run Lambda functions in your favorite programming language. For details, see Custom Lambda runtimes.

Application Load Balancer triggers

Elastic Load Balancing now supports Lambda functions as a target for Application Load Balancers. For details, see Using Lambda with application load balancers.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Kinesis HTTP/2 stream consumers as a trigger</td>
<td>You can use Kinesis HTTP/2 data stream consumers to send events to AWS Lambda. Stream consumers have dedicated read throughput from each shard in your data stream and use HTTP/2 to minimize latency. For details, see <a href="#">Using Lambda with Kinesis.</a></td>
<td>November 19, 2018</td>
</tr>
<tr>
<td>Python 3.7</td>
<td>AWS Lambda now supports Python 3.7 with a new runtime. For more information, see <a href="#">Building Lambda functions with Python.</a></td>
<td>November 19, 2018</td>
</tr>
<tr>
<td>Payload limit increase for asynchronous function invocation</td>
<td>The maximum payload size for asynchronous invocations increased from 128 KB to 256 KB, which matches the maximum message size from an Amazon SNS trigger. For details, see <a href="#">Lambda quotas.</a></td>
<td>November 16, 2018</td>
</tr>
<tr>
<td>AWS GovCloud (US-East) Region</td>
<td>AWS Lambda is now available in the AWS GovCloud (US-East) Region.</td>
<td>November 12, 2018</td>
</tr>
<tr>
<td>Moved AWS SAM topics to a separate Developer Guide</td>
<td>A number of topics were focused on building serverless applications using the AWS Serverless Application Model (AWS SAM). These topics have been moved to <a href="#">AWS Serverless Application Model developer guide.</a></td>
<td>October 25, 2018</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td><strong>View Lambda applications in the console</strong></td>
<td>You can view the status of your Lambda applications on the Applications page in the Lambda console. This page shows the status of the AWS CloudFormation stack. It includes links to pages where you can view more information about the resources in the stack. You can also view aggregate metrics for the application and create custom monitoring dashboards.</td>
<td>October 11, 2018</td>
</tr>
<tr>
<td><strong>Function execution timeout limit</strong></td>
<td>To allow for long-running functions, the maximum configurable execution timeout increased from 5 minutes to 15 minutes. For details, see Lambda limits.</td>
<td>October 10, 2018</td>
</tr>
<tr>
<td><strong>Support for PowerShell Core language in AWS Lambda</strong></td>
<td>AWS Lambda now supports the PowerShell Core language. For more information, see Programming model for authoring Lambda functions in PowerShell.</td>
<td>September 11, 2018</td>
</tr>
<tr>
<td><strong>Support for .NET Core 2.1.0 runtime in AWS Lambda</strong></td>
<td>AWS Lambda now supports the .NET Core 2.1.0 runtime. For more information, see .NET Core CLI.</td>
<td>July 9, 2018</td>
</tr>
<tr>
<td><strong>Updates now available over RSS</strong></td>
<td>You can now subscribe to an RSS feed to follow releases for this guide.</td>
<td>July 5, 2018</td>
</tr>
</tbody>
</table>
Support for Amazon SQS as event source

AWS Lambda now supports Amazon Simple Queue Service (Amazon SQS) as an event source. For more information, see Invoking Lambda functions.

China (Ningxia) Region

AWS Lambda is now available in the China (Ningxia) Region. For more information about Lambda Regions and endpoints, see Regions and endpoints in the AWS General Reference.

Earlier updates

The following table describes the important changes in each release of the AWS Lambda Developer Guide before June 2018.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime support for Node.js runtime 8.10</td>
<td>AWS Lambda now supports Node.js runtime version 8.10. For more information, see Building Lambda functions with Node.js.</td>
<td>April 2, 2018</td>
</tr>
<tr>
<td>Function and alias revision IDs</td>
<td>AWS Lambda now supports revision IDs on your function versions and aliases. You can use these IDs to track and apply conditional updates when you are updating your function version or alias resources.</td>
<td>January 25, 2018</td>
</tr>
<tr>
<td>Runtime support for Go and .NET 2.0</td>
<td>AWS Lambda has added runtime support for Go and .NET 2.0. For more information, see Building Lambda functions with Go and Building Lambda functions with C#.</td>
<td>January 15, 2018</td>
</tr>
<tr>
<td>Console Redesign</td>
<td>AWS Lambda has introduced a new Lambda console to simplify your experience and added a Cloud9 Code Editor</td>
<td>November 30, 2017</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>Change</td>
<td>to enhance your ability debug and revise your function code. For more information, see <a href="https://aws.amazon.com">Edit code using the console editor</a>.</td>
<td>November 30, 2017</td>
</tr>
<tr>
<td>Setting Concurrency Limits on Individual Functions</td>
<td>AWS Lambda now supports setting concurrency limits on individual functions. For more information, see <a href="https://aws.amazon.com">Configuring reserved concurrency</a>.</td>
<td>November 30, 2017</td>
</tr>
<tr>
<td>Shifting Traffic with Aliases</td>
<td>AWS Lambda now supports shifting traffic with aliases. For more information, see <a href="https://aws.amazon.com">Rolling deployments for Lambda functions</a>.</td>
<td>November 28, 2017</td>
</tr>
<tr>
<td>Gradual Code Deployment</td>
<td>AWS Lambda now supports safely deploying new versions of your Lambda function by leveraging Code Deploy. For more information, see <a href="https://aws.amazon.com">Gradual code deployment</a>.</td>
<td>November 28, 2017</td>
</tr>
<tr>
<td>China (Beijing) Region</td>
<td>AWS Lambda is now available in the China (Beijing) Region. For more information about Lambda regions and endpoints, see <a href="https://aws.amazon.com">Regions and endpoints</a> in the <a href="https://aws.amazon.com">AWS General Reference</a>.</td>
<td>November 9, 2017</td>
</tr>
<tr>
<td>Introducing SAM Local</td>
<td>AWS Lambda introduces SAM Local (now known as SAM CLI), a AWS CLI tool that provides an environment for you to develop, test, and analyze your serverless applications locally before uploading them to the Lambda runtime. For more information, see <a href="https://aws.amazon.com">Testing and debugging serverless applications</a>.</td>
<td>August 11, 2017</td>
</tr>
<tr>
<td>Canada (Central) Region</td>
<td>AWS Lambda is now available in the Canada (Central) Region. For more information about Lambda regions and endpoints, see <a href="https://aws.amazon.com">Regions and endpoints</a> in the <a href="https://aws.amazon.com">AWS General Reference</a>.</td>
<td>June 22, 2017</td>
</tr>
<tr>
<td>South America (São Paulo) Region</td>
<td>AWS Lambda is now available in the South America (São Paulo) Region. For more information about Lambda regions and endpoints, see <a href="https://aws.amazon.com">Regions and endpoints</a> in the <a href="https://aws.amazon.com">AWS General Reference</a>.</td>
<td>June 6, 2017</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>-----------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>AWS Lambda support for AWS X-Ray.</td>
<td>Lambda introduces support for X-Ray, which allows you to detect, analyze, and optimize performance issues with your Lambda applications. For more information, see Using AWS Lambda with AWS X-Ray.</td>
<td>April 19, 2017</td>
</tr>
<tr>
<td>Asia Pacific (Mumbai) Region</td>
<td>AWS Lambda is now available in the Asia Pacific (Mumbai) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the AWS General Reference.</td>
<td>March 28, 2017</td>
</tr>
<tr>
<td>AWS Lambda now supports Node.js runtime v6.10</td>
<td>AWS Lambda added support for Node.js runtime v6.10. For more information, see Building Lambda functions with Node.js.</td>
<td>March 22, 2017</td>
</tr>
<tr>
<td>Europe (London) Region</td>
<td>AWS Lambda is now available in the Europe (London) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the AWS General Reference.</td>
<td>February 1, 2017</td>
</tr>
<tr>
<td>AWS Lambda support for the .NET runtime, Lambda@Edge (Preview), Dead Letter Queues and automated deployment of serverless applications.</td>
<td>AWS Lambda added support for C#. For more information, see Building Lambda functions with C#. Lambda@Edge allows you to run Lambda functions at the AWS Edge locations in response to CloudFront events. For more information, see Using AWS Lambda with CloudFront Lambda@Edge.</td>
<td>December 3, 2016</td>
</tr>
<tr>
<td>AWS Lambda adds Amazon Lex as a supported event source.</td>
<td>Using Lambda and Amazon Lex, you can quickly build chat bots for various services like Slack and Facebook. For more information, see Using AWS Lambda with Amazon Lex.</td>
<td>November 30, 2016</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>US West (N. California) Region</td>
<td>AWS Lambda is now available in the US West (N. California) Region. For more information about Lambda regions and endpoints, see <a href="#">Regions and endpoints</a> in the <a href="#">AWS General Reference</a>.</td>
<td>November 21, 2016</td>
</tr>
<tr>
<td>Introduced the AWS SAM for creating and deploying Lambda-based applications and using environment variables for Lambda function configuration settings.</td>
<td>AWS SAM: You can now use the AWS SAM to define the syntax for expressing resources within a serverless application. In order to deploy your application, simply specify the resources you need as part of your application, along with their associated permissions policies in a AWS CloudFormation template file (written in either JSON or YAML), package your deployment artifacts, and deploy the template. For more information, see <a href="#">AWS Lambda applications</a>. Environment variables: You can use environment variables to specify configuration settings for your Lambda function outside of your function code. For more information, see <a href="#">Using Lambda environment variables</a>.</td>
<td>November 18, 2016</td>
</tr>
<tr>
<td>Asia Pacific (Seoul) Region</td>
<td>AWS Lambda is now available in the Asia Pacific (Seoul) Region. For more information about Lambda regions and endpoints, see <a href="#">Regions and endpoints</a> in the <a href="#">AWS General Reference</a>.</td>
<td>August 29, 2016</td>
</tr>
<tr>
<td>Asia Pacific (Sydney) Region</td>
<td>Lambda is now available in the Asia Pacific (Sydney) Region. For more information about Lambda regions and endpoints, see <a href="#">Regions and endpoints</a> in the <a href="#">AWS General Reference</a>.</td>
<td>June 23, 2016</td>
</tr>
<tr>
<td>Updates to the Lambda console</td>
<td>The Lambda console has been updated to simplify the role-creation process.</td>
<td>June 23, 2016</td>
</tr>
<tr>
<td>AWS Lambda now supports Node.js runtime v4.3</td>
<td>AWS Lambda added support for Node.js runtime v4.3. For more information, see <a href="#">Building Lambda functions with Node.js</a>.</td>
<td>April 07, 2016</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>Europe (Frankfurt) region</td>
<td>Lambda is now available in the Europe (Frankfurt) region. For more information about Lambda regions and endpoints, see <a href="#">Regions and endpoints</a> in the AWS General Reference.</td>
<td>March 14, 2016</td>
</tr>
<tr>
<td>VPC support</td>
<td>You can now configure a Lambda function to access resources in your VPC. For more information, see <a href="#">Connecting outbound networking to resources in a VPC</a>.</td>
<td>February 11, 2016</td>
</tr>
<tr>
<td>Lambda runtime has been updated.</td>
<td>The execution environment has been updated.</td>
<td>November 4, 2015</td>
</tr>
<tr>
<td>Versioning support, Python for developing code for Lambda functions, scheduled events, and increase in execution time</td>
<td>You can now develop your Lambda function code using Python. For more information, see <a href="#">Building Lambda functions with Python</a>. Versioning: You can maintain one or more versions of your Lambda function. Versioning allows you to control which Lambda function version is executed in different environments (for example, development, testing, or production). For more information, see <a href="#">Lambda function versions</a>. Scheduled events: You can also set up Lambda to invoke your code on a regular, scheduled basis using the Lambda console. You can specify a fixed rate (number of hours, days, or weeks) or you can specify a cron expression. For an example, see <a href="#">Using AWS Lambda with Amazon EventBridge (CloudWatch Events)</a>. Increase in execution time: You can now set up your Lambda functions to run for up to five minutes allowing longer running functions such as large volume data ingestion and processing jobs.</td>
<td>October 8, 2015</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>Support for DynamoDB Streams</td>
<td>DynamoDB Streams is now generally available and you can use it in all the regions where DynamoDB is available. You can enable DynamoDB Streams for your table and use a Lambda function as a trigger for the table. Triggers are custom actions you take in response to updates made to the DynamoDB table. For an example walkthrough, see <a href="https://aws.amazon.com/">Tutorial: Using AWS Lambda with Amazon DynamoDB streams</a>.</td>
<td>July 14, 2015</td>
</tr>
<tr>
<td>Lambda now supports invoking Lambda functions with REST-compatible clients.</td>
<td>Until now, to invoke your Lambda function from your web, mobile, or IoT application you needed the AWS SDKs (for example, AWS SDK for Java, AWS SDK for Android, or AWS SDK for iOS). Now, Lambda supports invoking a Lambda function with REST-compatible clients through a customized API that you can create using Amazon API Gateway. You can send requests to your Lambda function endpoint URL. You can configure security on the endpoint to allow open access, leverage AWS Identity and Access Management (IAM) to authorize access, or use API keys to meter access to your Lambda functions by others. For an example Getting Started exercise, see <a href="https://aws.amazon.com/">Using AWS Lambda with Amazon API Gateway</a>. For more information about the Amazon API Gateway, see <a href="https://aws.amazon.com/api-gateway/">https://aws.amazon.com/api-gateway/</a>.</td>
<td>July 09, 2015</td>
</tr>
<tr>
<td>The Lambda console now provides blueprints to easily create Lambda functions and test them.</td>
<td>Lambda console provides a set of blueprints. Each blueprint provides a sample event source configuration and sample code for your Lambda function that you can use to easily create Lambda-based applications. All of the Lambda Getting Started exercises now use the blueprints. For more information, see <a href="https://aws.amazon.com/">Getting started with Lambda</a>.</td>
<td>July 09, 2015</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>Lambda now supports Java to author your Lambda functions.</td>
<td>You can now author Lambda code in Java. For more information, see Building Lambda functions with Java.</td>
<td>June 15, 2015</td>
</tr>
<tr>
<td>Lambda now supports specifying an Amazon S3 object as the function .zip when creating or updating a Lambda function.</td>
<td>You can upload a Lambda function deployment package (.zip file) to an Amazon S3 bucket in the same region where you want to create a Lambda function. Then, you can specify the bucket name and object key name when you create or update a Lambda function.</td>
<td>May 28, 2015</td>
</tr>
<tr>
<td>Lambda now generally available with added support for mobile backends</td>
<td>Lambda is now generally available for production use. The release also introduces new features that make it easier to build mobile, tablet, and Internet of Things (IoT) backends using Lambda that scale automatically without provisioning or managing infrastructure. Lambda now supports both real-time (synchronous) and asynchronous events. Additional features include easier event source configuration and management. The permission model and the programming model have been simplified by the introduction of resource policies for your Lambda functions. The documentation has been updated accordingly. For information, see the following topics: Getting started with Lambda AWS Lambda</td>
<td>April 9, 2015</td>
</tr>
<tr>
<td>Preview release</td>
<td>Preview release of the AWS Lambda Developer Guide.</td>
<td>November 13, 2014</td>
</tr>
</tbody>
</table>