Developer Guide

AWS X-Ray

AWS X-Ray: Developer Guide

Table of Contents

What is AWS X-Ray? 1
Getting started 4
Choosing an interface 6
Use an SDK 8
Use the ADOT SDK 9
Use the X-Ray SDK 10
Use a console 12
Use the Amazon CloudWatch console 12
Use the X-Ray console 13
Explore the X-Ray console 14
Trace map 15
Traces 22
Filter expressions 30
Cross-account tracing 42
Tracing event-driven applications 46
Histograms 49
Insights 52
Analytics 59
Groups 67
Sampling 76
Console deep linking 83
Use the X-Ray API 85
Tutorial 87
Sending data 92
Getting data 97
Configuration 111
Sampling 118
Segment documents 122
Concepts 142
Segments 142
Subsegments 143
Service graph 147
Traces 148
Sampling 149

Tracing header ... 150
Filter expressions ... 151
Groups ... 152
Annotations and metadata ... 152
Errors, faults, and exceptions ... 153
Security ... 154
... 154
Data protection ... 154
Identity and access management ... 157
Audience ... 157
Authenticating with identities ... 158
Managing access using policies ... 161
How AWS X-Ray works with IAM ... 163
Identity-based policy examples ... 170
Troubleshooting ... 181
Logging and monitoring ... 183
Compliance validation ... 184
Resilience ... 185
Infrastructure security ... 185
VPC endpoints ... 186
Creating a VPC endpoint for X-Ray ... 186
Controlling access to your X-Ray VPC endpoint ... 188
Supported Regions ... 189
Cross-service confused deputy prevention ... 190
Sample application ... 192
Scorekeep tutorial ... 194
Prerequisites ... 195
Install the Scorekeep application using CloudFormation ... 196
Generate trace data ... 197
View the trace map in the AWS Management Console ... 198
Configuring Amazon SNS notifications ... 206
Explore the sample application ... 207
Optional: Least privilege policy ... 212
Clean up ... 214
Next steps ... 215
AWS SDK clients ... 216

- Custom subsegments 217
- Annotations and metadata 217
- HTTP clients 219
- SQL clients 220
- AWS Lambda functions 223
  - Random name 224
  - Worker 226
- Instrumenting startup code 228
- Instrumenting scripts 231
- Instrumenting web clients 233
- Worker threads 237

X-Ray daemon 239

- Downloading the daemon 240
- Verifying the daemon archive's signature 241
- Running the daemon 242
- Giving the daemon permission to send data to X-Ray 243
- X-Ray daemon logs 243
- Configuration 244
  - Supported environment variables 245
  - Using command line options 245
  - Using a configuration file 247
- Run the daemon locally 248
  - Running the X-Ray daemon on Linux 249
  - Running the X-Ray daemon in a Docker container 249
  - Running the X-Ray daemon on Windows 250
  - Running the X-Ray daemon on OS X 251
- On Elastic Beanstalk 252
  - Using the Elastic Beanstalk X-Ray integration to run the X-Ray daemon 253
  - Downloading and running the X-Ray daemon manually (advanced) 254
- On Amazon EC2 256
- On Amazon ECS 258
  - Using the official Docker image 258
  - Create and build a Docker image 259
  - Configure command line options in the Amazon ECS console 262

Integrating with AWS services 264

- Amazon Bedrock AgentCore 266

- Amazon S3 267
- Amazon S3 267
- Amazon EC2 267
- Amazon SNS 267
  - Configure Amazon SNS active tracing 268
  - View Amazon SNS publisher and subscriber traces in the X-Ray console 269
- Amazon SQS 271
  - Send the HTTP trace header 272
  - Retrieve the trace header and recover trace context 273
- Amazon S3 274
  - Configure Amazon S3 event notifications 274
- AWS Distro for OpenTelemetry 275
  - AWS Distro for OpenTelemetry 275
- AWS Config 276
  - Creating a Lambda function trigger 277
  - Creating a custom AWS Config rule for x-ray 278
  - Example results 279
  - Amazon SNS notifications 280
- AWS AppSync 280
- API Gateway 280
- App Mesh 282
- App Runner 285
- CloudTrail 285
  - X-Ray management events in CloudTrail 287
  - X-Ray data events in CloudTrail 287
  - X-Ray event examples 289
- CloudWatch 291
  - CloudWatch RUM 292
  - CloudWatch Synthetics 293
- Elastic Beanstalk 302
- Elastic Load Balancing 303
- EventBridge 304
  - Viewing source and targets on the X-Ray service map 304
  - Propagate the trace context to event targets 304
- Lambda 311
- Step Functions 313

Instrumenting your application 315
Instrumenting your application with the AWS Distro for OpenTelemetry 316
Instrumenting your application with AWS X-Ray SDKs 317
Choosing between the AWS Distro for OpenTelemetry and X-Ray SDKs 318
Transaction Search 320
OpenTelemetry Protocol (OTLP) Endpoint 321
Working with Go 322
AWS Distro for OpenTelemetry Go 322
X-Ray SDK for Go 323
Requirements 325
Reference documentation 325
Configuration 325
Incoming requests 332
AWS SDK clients 335
Outgoing HTTP calls 337
SQL queries 338
Custom subsegments 339
Annotations and metadata 340
Working with Java 343
AWS Distro for OpenTelemetry Java 343
X-Ray SDK for Java 344
Submodules 345
Requirements 346
Dependency management 347
Auto-instrumentation agent 349
Configuration 360
Incoming requests 371
AWS SDK clients 376
Outgoing HTTP calls 378
SQL queries 381
Custom subsegments 384
Annotations and metadata 386
Monitoring 392
Multithreading 396
AOP with Spring 397
Working with Node.js 403

AWS Distro for OpenTelemetry JavaScript 403
X-Ray SDK for Node.js 404
Requirements 406
Dependency management 406
Node.js samples 407
Configuration 408
Incoming requests 413
AWS SDK clients 417
Outgoing HTTP calls 421
SQL queries 423
Custom subsegments 425
Annotations and metadata 427
Working with Python 432
AWS Distro for OpenTelemetry Python 432
X-Ray SDK for Python 433
Requirements 435
Dependency management 436
Configuration 436
Incoming requests 443
Patching libraries 450
AWS SDK clients 453
Outgoing HTTP calls 454
Custom subsegments 456
Annotations and metadata 458
Instrument serverless applications 462
Working with .NET 469
AWS Distro for OpenTelemetry .NET 469
X-Ray SDK for .NET 470
Requirements 472
Adding the X-Ray SDK for .NET to your application 472
Dependency management 472
Configuration 474
Incoming requests 481
AWS SDK clients 485
Outgoing HTTP calls 488
SQL queries 490

Custom subsegments ... 493
Annotations and metadata ... 495
Working with Ruby ... 499
AWS Distro for OpenTelemetry Ruby ... 499
X-Ray SDK for Ruby ... 500
Requirements ... 501
Configuration ... 502
Incoming requests ... 508
Patching libraries ... 512
AWS SDK clients ... 513
Custom subsegments ... 515
Annotations and metadata ... 516
X-Ray SDK and daemon end of support timeline ... 520
Migrating from X-Ray instrumentation to OpenTelemetry instrumentation ... 522
Understanding OpenTelemetry ... 523
OpenTelemetry support in AWS ... 523
Understanding OpenTelemetry concepts for migration ... 524
Comparing features ... 525
Setting up and configuring tracing ... 526
Detecting resources in your environment ... 527
Managing sampling strategies ... 528
Managing trace context ... 528
Propagating trace context ... 529
Using library instrumentation ... 529
Exporting traces ... 530
Processing and forwarding traces ... 530
Span processing (OpenTelemetry-specific concept) ... 531
Baggage (OpenTelemetry-soecific concept) ... 531
Migration overview ... 531
Recommendations for new and existing applications ... 532
Tracing setup changes ... 533
Library instrumentation changes ... 533
Lambda environment instrumentation changes ... 533
Manually creating trace data ... 534
Migrating from X-Ray Daemon to AWS CloudWatch agent or OpenTelemetry collector ... 535
Migrating on Amazon EC2 or on-premises servers ... 535

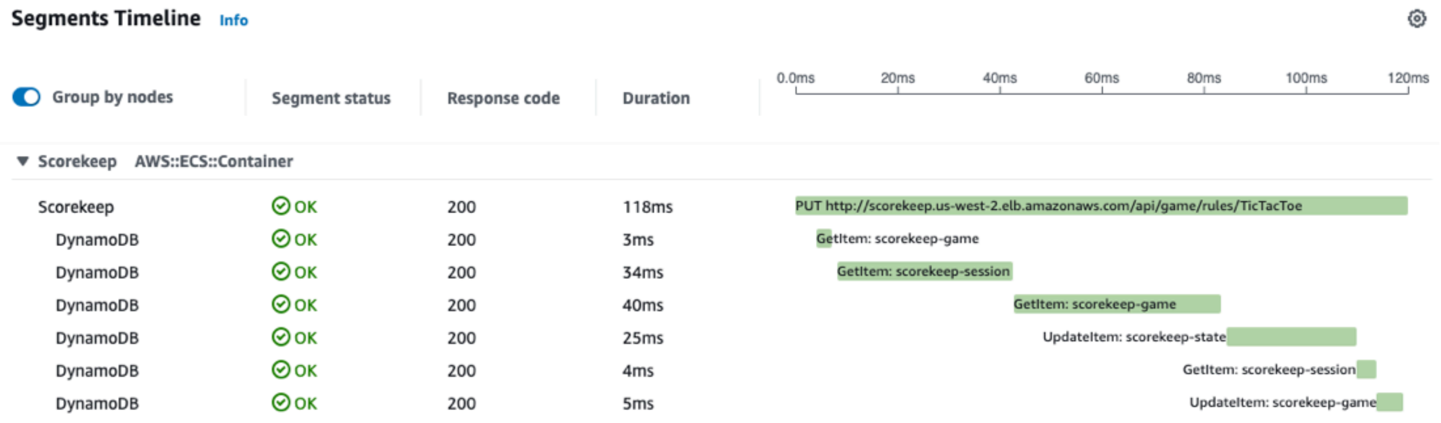
Migrating on Amazon ECS 539
Migrating on Elastic Beanstalk 543
Migrating to OpenTelemetry Java 544
Zero code automatic instrumentation solution 545
Manual instrumentation solutions with the SDK 545
Tracing incoming requests (spring framework instrumentation) 548
AWS SDK v2 instrumentation 549
Instrumenting outgoing HTTP calls 551
Instrumentation support for other libraries 552
Manually creating trace data 552
Lambda instrumentation 555
Migrate to OpenTelemetry Go 561
Manual instrumentation with the SDK 561
Tracing incoming requests (HTTP handler instrumentation) 563
AWS SDK for Go v2 instrumentation 564
Instrumenting outgoing HTTP calls 565
Instrumentation support for other libraries 566
Manually creating trace data 567
Lambda manual instrumentation 568
Migrate to OpenTelemetry Node.js 575
Zero code automatic instrumentation solutions 575
Manual instrumentation solutions 576
Tracing incoming requests 579
AWS SDK JavaScript V3 instrumentation 564
Instrumenting outgoing HTTP calls 582
Instrumentation support for other libraries 583
Manually creating trace data 567
Lambda instrumentation 568
Migrate to OpenTelemetry .NET 587
Zero code automatic instrumentation solutions 587
Manual instrumentation solutions with the SDK 588
Manually creating trace data 591
Tracing incoming requests (ASP.NET and ASP.NET core instrumentation) 594
AWS SDK instrumentation 595
Instrumenting outgoing HTTP calls 596
Instrumentation support for other libraries 597

Lambda instrumentation 568
Migrate to OpenTelemetry Python 601
Zero code automatic instrumentation solutions 602
Manually instrument your applications 602
Tracing setup initialization 603
Tracing incoming requests 606
AWS SDK instrumentation 607
Instrumenting outgoing HTTP calls through requests 609
Instrumentation support for other libraries 610
Manually creating trace data 610
Lambda instrumentation 612
Migrate to OpenTelemetry Ruby 613
Manually instrument your solutions with the SDK 614
Tracing incoming requests (Rails instrumentation) 616
AWS SDK instrumentation 617
Instrumenting outgoing HTTP calls 618
Instrumentation support for other libraries 619
Manually creating trace data 619
Lambda manual instrumentation 622
Creating X-Ray resources with CloudFormation 625
X-Ray and AWS CloudFormation templates 625
Learn more about AWS CloudFormation 625
Tagging 626
Tag restrictions 627
Managing tags in the console 627
Add tags to a new group (console) 628
Add tags to a new sampling rule (console) 628
Edit or delete tags for a group (console) 629
Edit or delete tags for a sampling rule (console) 629
Managing tags in the AWS CLI 629
Add tags to a new X-Ray group or sampling rule (CLI) 630
Add tags to an existing resource (CLI) 632
List tags on a resource (CLI) 633
Delete tags on a resource (CLI) 633
Control access to X-Ray resources based on tags 634
Troubleshooting 635

X-Ray trace map and trace details pages 635
I don't see all of my CloudWatch logs 635
I don't see all of my alarms on the X-Ray trace map 636
I don't see some AWS resources on the trace map 636
There are too many nodes on the trace map 637
X-Ray SDK for Java 637
X-Ray SDK for Node.js 637
The X-Ray daemon 638
Document History 639

What is AWS X-Ray?

AWS X-Ray is a service that collects data about requests that your application serves, and provides tools that you can use to view, filter, and gain insights into that data to identify issues and opportunities for optimization. For any traced request to your application, you can see detailed information not only about the request and response, but also about calls that your application makes to downstream AWS resources, microservices, databases, and web APIs.



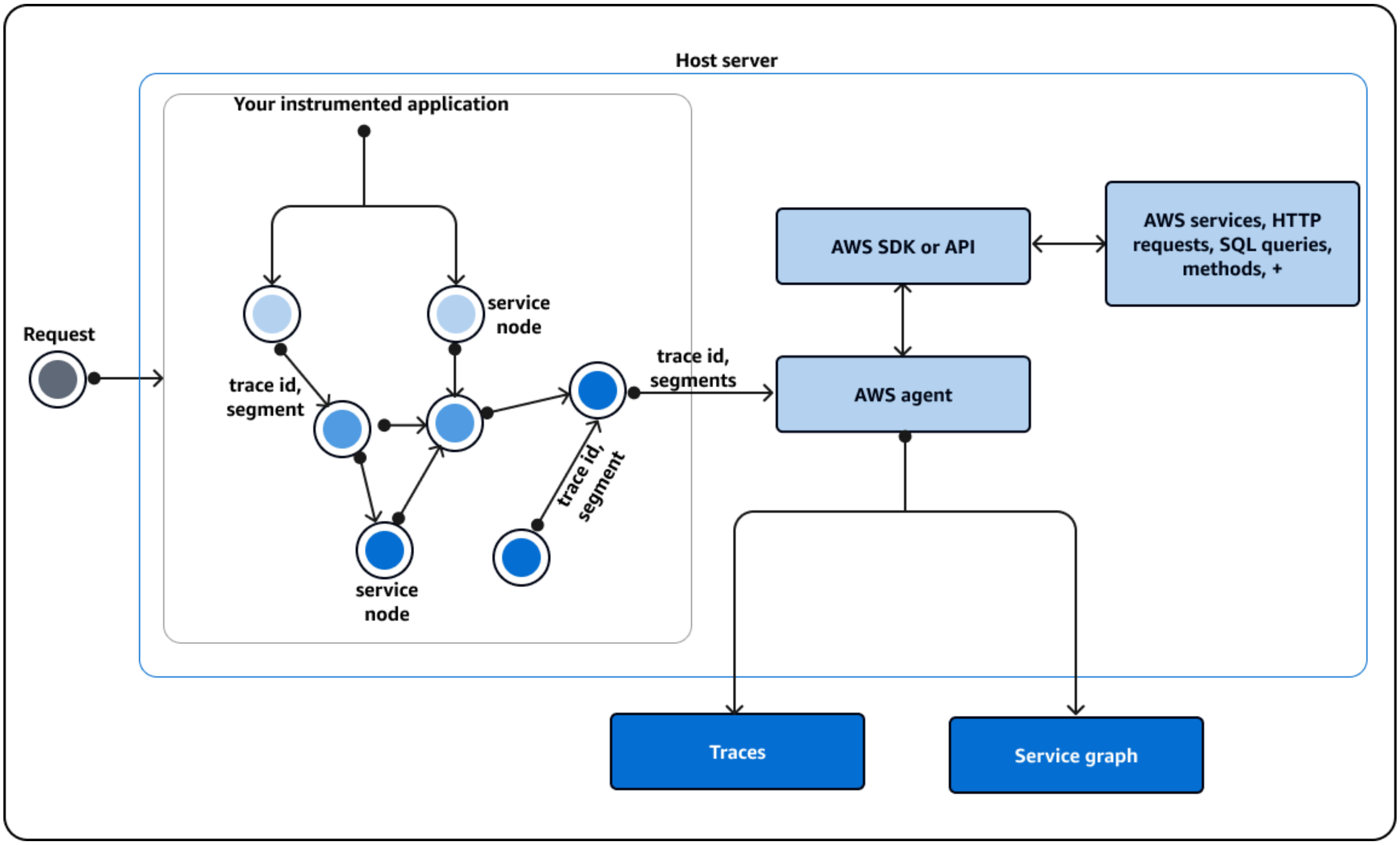
Segments Timeline Info

▼ Scorekeep AWS::ECS::Container

Group by nodes	Segment status	Response code	Duration	
Scorekeep	✓ OK	200	118ms	PUT http://scorekeep.us-west-2.elb.amazonaws.com/api/game/rules/TicTacToe
DynamoDB	✓ OK	200	3ms	GetItem: scorekeep-game
DynamoDB	✓ OK	200	34ms	GetItem: scorekeep-session
DynamoDB	✓ OK	200	40ms	GetItem: scorekeep-game
DynamoDB	✓ OK	200	25ms	UpdateItem: scorekeep-state
DynamoDB	✓ OK	200	4ms	GetItem: scorekeep-session
DynamoDB	✓ OK	200	5ms	UpdateItem: scorekeep-game

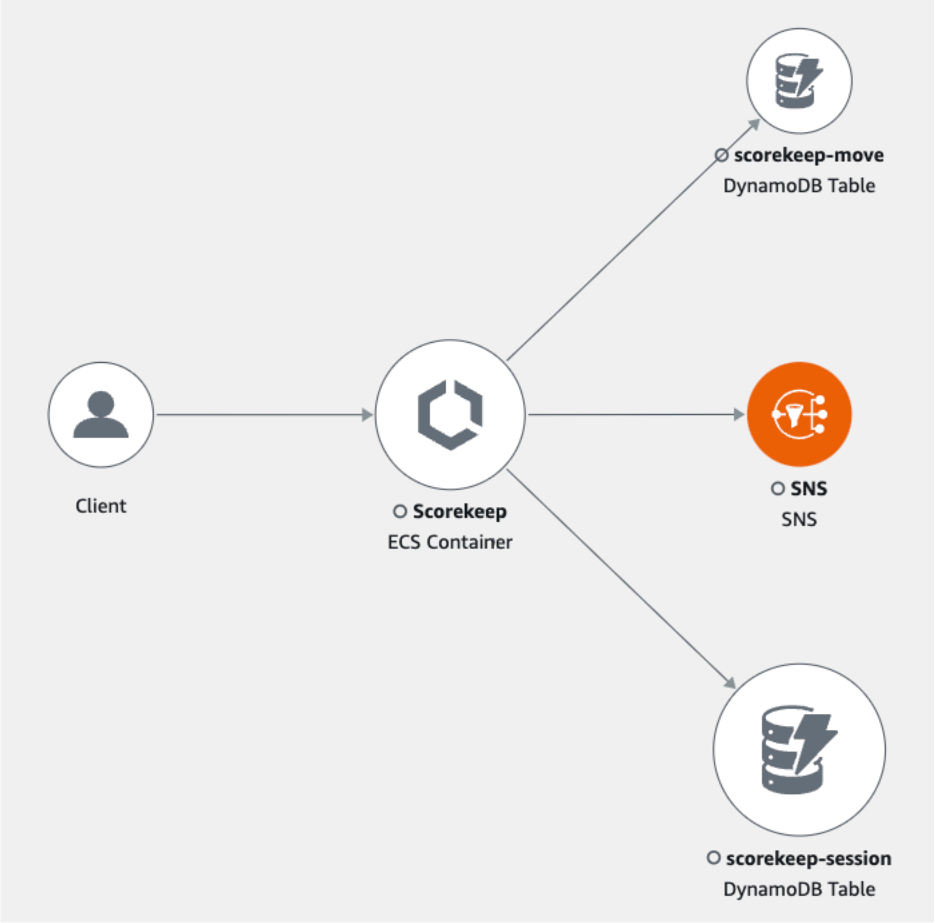
AWS X-Ray receives traces from your application, in addition to AWS services your application uses that are already integrated with X-Ray. Instrumenting your application involves sending trace data for incoming and outbound requests and other events within your application, along with metadata about each request. Many instrumentation scenarios require only configuration changes. For example, you can instrument all incoming HTTP requests and downstream calls to AWS services that your Java application makes. There are several SDKs, agents, and tools that can be used to instrument your application for X-Ray tracing. See Instrumenting your application for more information.

AWS services that are integrated with X-Ray can add tracing headers to incoming requests, send trace data to X-Ray, or run the X-Ray daemon. For example, AWS Lambda can send trace data about requests to your Lambda functions, and run the X-Ray daemon on workers to make it simpler to use the X-Ray SDK.



Instead of sending trace data directly to X-Ray, each client SDK sends JSON segment documents to a daemon process listening for UDP traffic. The X-Ray daemon buffers segments in a queue and uploads them to X-Ray in batches. The daemon is available for Linux, Windows, and macOS, and is included on AWS Elastic Beanstalk and AWS Lambda platforms.

X-Ray uses trace data from the AWS resources that power your cloud applications to generate a detailed *trace map*. The trace map shows the client, your front-end service, and backend services that your front-end service calls to process requests and persist data. Use the trace map to identify bottlenecks, latency spikes, and other issues to solve to improve the performance of your applications.


scorekeep-move
DynamoDB Table
Client
Scorekeep
ECS Container
SNS
SNS
scorekeep-session
DynamoDB Table

Getting started with X-Ray

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

To use X-Ray, take the following steps:

1. Instrument your application, which allows X-Ray to track how your application processes a request.

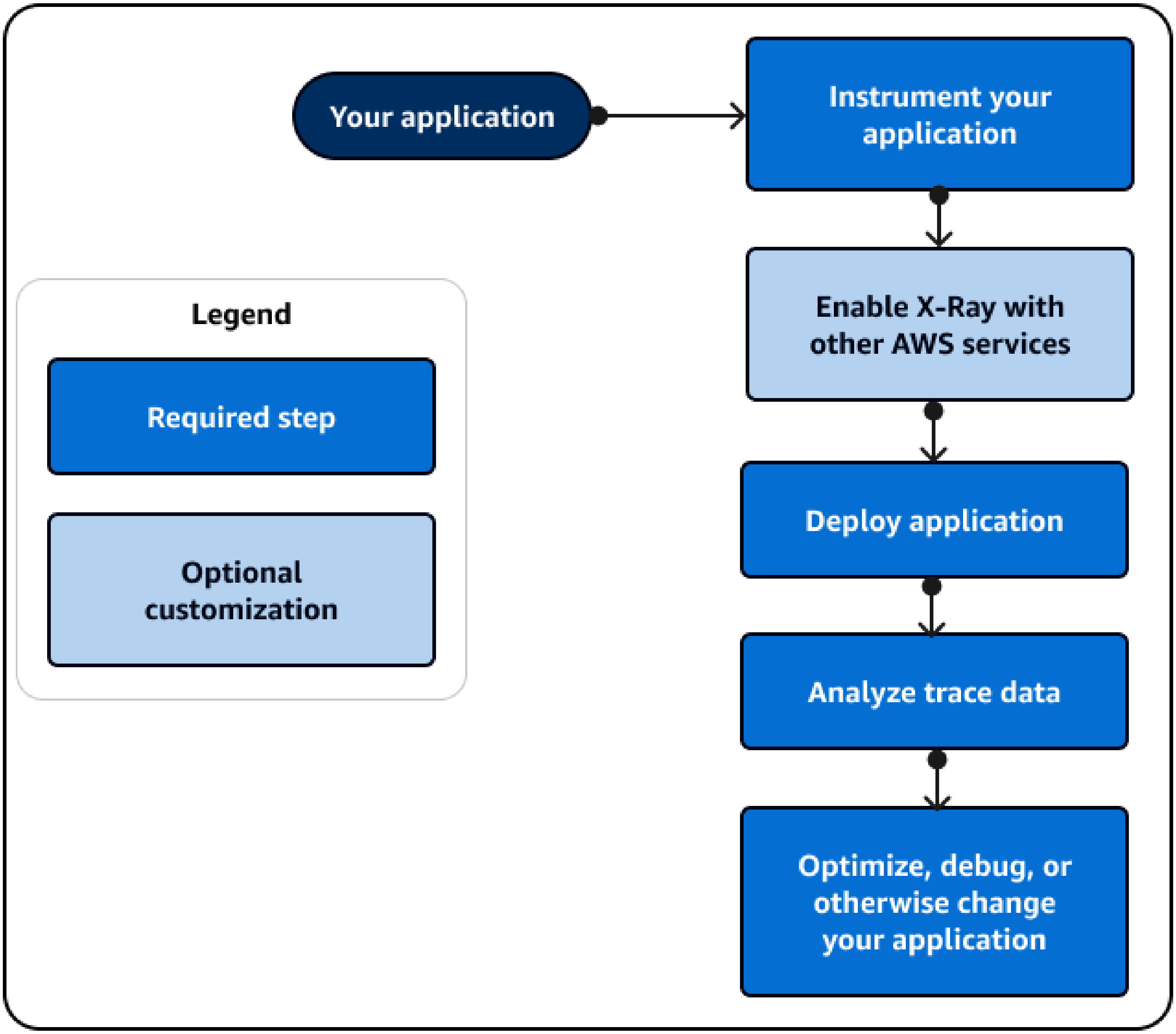
   - Use the X-Ray SDKs, X-Ray APIs, ADOT or CloudWatch Application Signals to send trace data to X-Ray. For more information about which interface to use, see Choosing an interface.

   For more information about instrumentation, see Instrumenting your application for AWS X-Ray.
2. (Optional) Configure X-Ray to work with other AWS services that integrate with X-Ray. You can sample traces and add headers to incoming requests, run an agent or collector, and automatically send trace data to X-Ray. For more information, see Integrating AWS X-Ray with other AWS services.
3. Deploy your instrumented application. As your application receives requests, the X-Ray SDK will record trace, segment and subsegment data. In this step, you might also have to set up an IAM policy and deploy an agent or collector.

   - For example scripts to deploy an application using the AWS Distro for OpenTelemetry (ADOT) SDK and the CloudWatch agent on different platforms, see Application Signals Demo Scripts.
   - For an example script to deploy an application using the X-Ray SDK and the X-Ray daemon, see AWS X-Ray sample application.

4. (Optional) Open a console to view and analyze the data. You can see a GUI representation of a trace map, service map, and more to inspect how your application functions. Use the graphical information in the console to optimize, debug and understand your application. For more information about choosing a console, see Use a console.

The following diagram shows how to get started using X-Ray:


Your application
Instrument your application
Enable X-Ray with other AWS services
Deploy application
Analyze trace data
Optimize, debug, or otherwise change your application
Legend
Required step
Optional customization

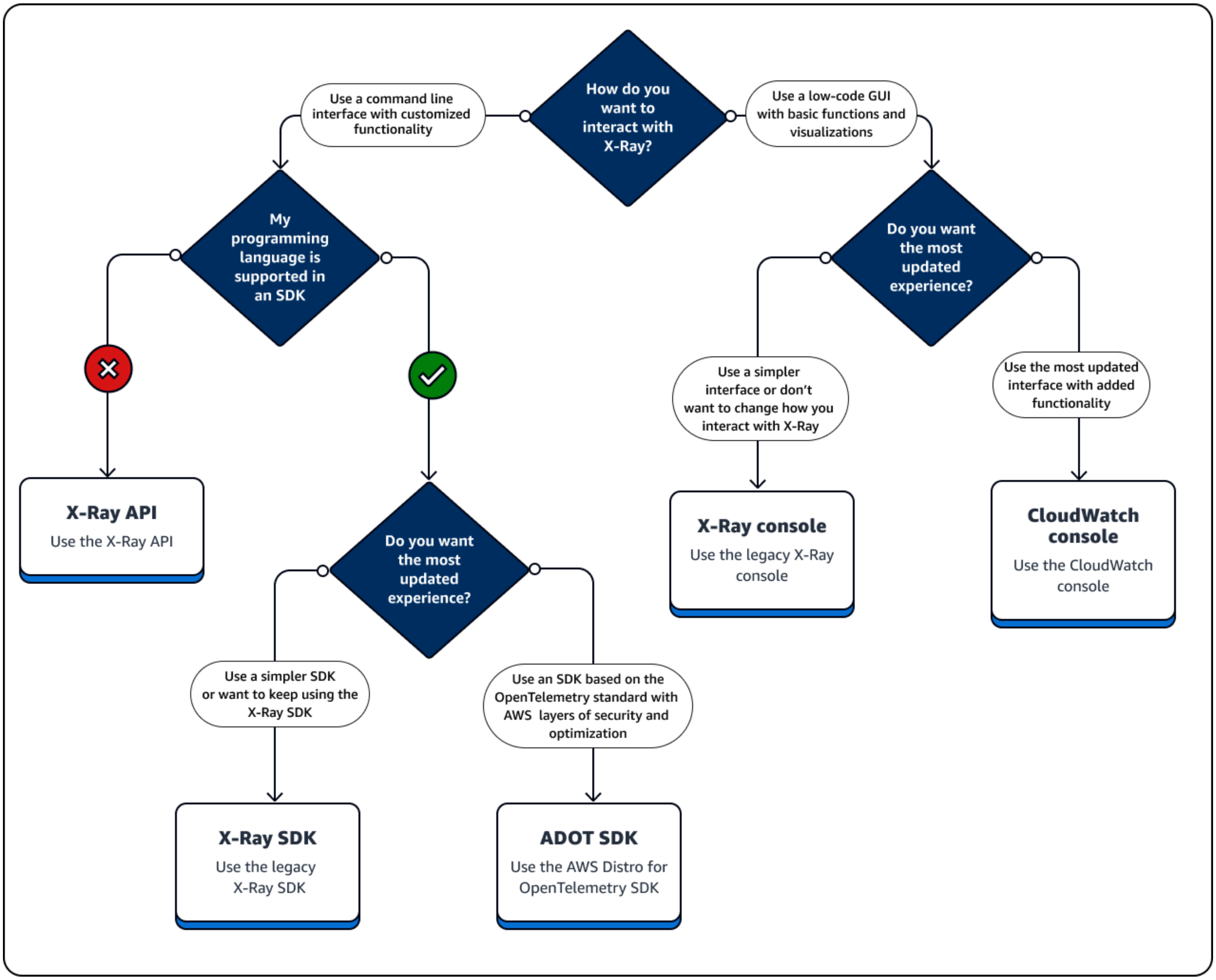
For an example of the data and maps that are available in the console, launch a sample application that is already instrumented to generate trace data. In a few minutes, you can generate traffic, send segments to X-Ray, and view a trace and service map.

Choosing an interface

AWS X-Ray can provide insights into how your application works and how well it interacts with other services and resources. After you instrument or configure your application, X-Ray collects trace data as your application serves requests. You can analyze this trace data to identify performance issues, troubleshoot errors, and optimization your resources. This guide shows you how to interact with X-Ray with the following guidelines:

- Use an AWS Management Console if you want to get started quickly or can use pre-built visualizations to perform basic tasks.
  - Choose the Amazon CloudWatch console for the most updated user experience that contains all of the X-Ray console's functionality.
  - Use the X-Ray console if you want a simpler interface or don't want to change how you interact with X-Ray.
- Use an SDK if you need more custom tracing, monitoring or logging capabilities than an AWS Management Console can provide.
  - Choose the ADOT SDK if you want a vendor-agnostic SDK based on the open source OpenTelemetry SDK with added layers of AWS security and optimization.
  - Choose the X-Ray SDK if you want a simpler SDK or don't want to update your application code.
- Use X-Ray API operations if an SDK does not support your application's programming language.

The following diagram helps you choose how to interact with X-Ray:


How do you want to interact with X-Ray?
Use a command line interface with customized functionality
Use a low-code GUI with basic functions and visualizations
My programming language is supported in an SDK
Do you want the most updated experience?
X-Ray API
Use the X-Ray API
Do you want the most updated experience?
Use a simpler SDK or want to keep using the X-Ray SDK
Use an SDK based on the OpenTelemetry standard with AWS layers of security and optimization
X-Ray SDK
Use the legacy X-Ray SDK
ADOT SDK
Use the AWS Distro for OpenTelemetry SDK
Use a simpler interface or don't want to change how you interact with X-Ray
Use the most updated interface with added functionality
X-Ray console
Use the legacy X-Ray console
CloudWatch console
Use the CloudWatch console

Explore the interface types

- Use an SDK
- Use a console
- Use the X-Ray API

Use an SDK

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

Use an SDK if you want to use a command line interface or need more custom tracing, monitoring, or logging capabilities than what is available in an AWS Management Console. You can also use an AWS SDK to develop programs that use the X-Ray APIs. You can use either the AWS Distro for OpenTelemetry (ADOT) SDK or the X-Ray SDK.

If you use an SDK, you can add customizations to your workflow both when you instrument your application and when you configure your collector or agent. You can use an SDK to do the following tasks that you can't do using an AWS Management Console:

- Publish custom metrics – Sample metrics at high resolutions down to 1 second, use multiple dimensions to add information about a metric, and aggregate data points into a statistic set.
- Customize your collector – Customize the configuration for any portion of a collector including the receiver, processor, exporter, and connector.
- Customize your instrumentation – Customize segments and subsegments, add custom key-value pairs as attributes, and create custom metrics.
- Create and update sampling rules programmatically.

Use the ADOT SDK if you want the flexibility of using a standardized OpenTelemetry SDK with added layers of AWS security and optimization. The AWS Distro for OpenTelemetry (ADOT) SDK is a vendor-agnostic package that allows for integration with back ends from other vendors and non-AWS services without having to reinstrument your code.

Use the X-Ray SDK if you are already using the X-Ray SDK, only integrate with AWS backends, and don't want to change the way you interact with X-Ray or your application code.

For more information about each feature, see Choosing between the AWS Distro for OpenTelemetry and X-Ray SDKs.

Use the ADOT SDK

The ADOT SDK is a set of open source APIs, libraries and agents that send data to backend services. ADOT is supported by AWS, integrates with multiple backends and agents, and provides a large number of open source libraries maintained by the OpenTelemetry community. Use the ADOT SDK to instrument your application and collect logs, metadata, metrics and traces. You can also use ADOT to monitor services and set an alarm based on your metrics in CloudWatch.

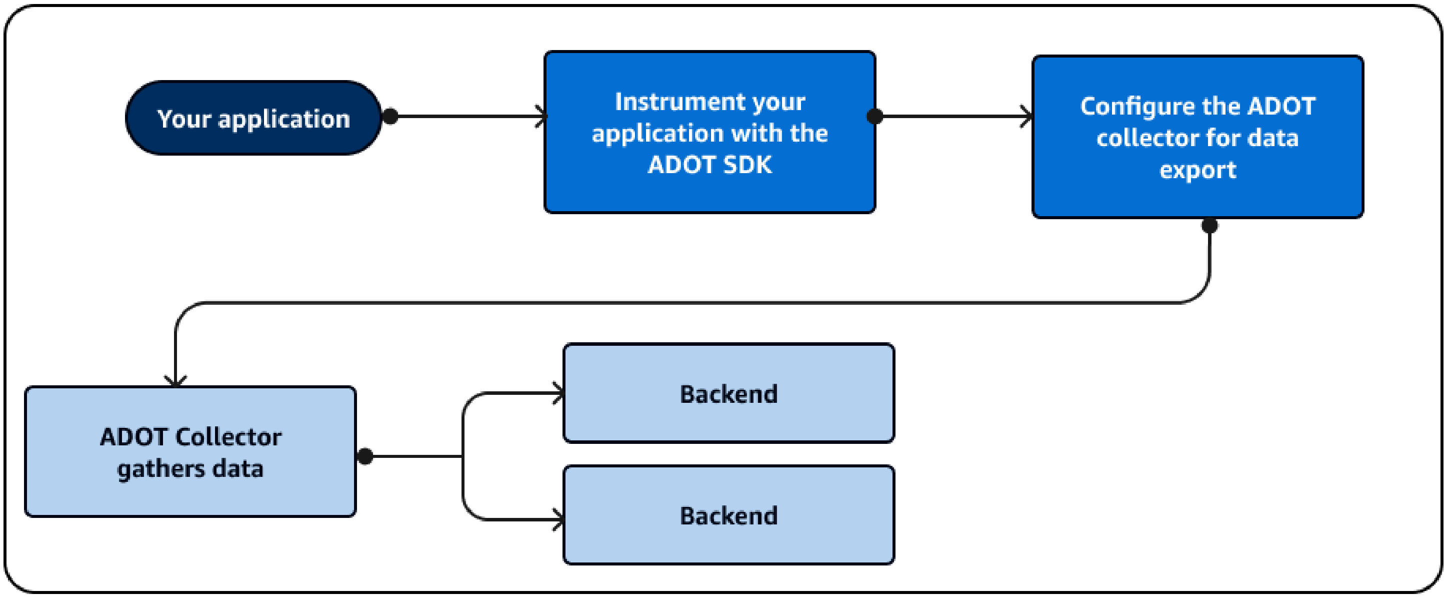
If you are using the ADOT SDK, you have the following options, in combination with an agent:

- Use the ADOT SDK with the CloudWatch agent – recommended.
- Use the ADOT SDK with the ADOT Collector – recommended if you want to use vendor agnostic software with AWS layers of security and optimization.

To use the ADOT SDK, do the following:

- Instrument your application using the ADOT SDK. For more information, see the documentation for your programming language in the ADOT technical documentation.
- Configure an ADOT collector to tell it where to send data that it collects.

After the ADOT collector receives your data, it sends it to the backend that you specify in the ADOT configuration. ADOT can send data to multiple backends, including to vendors outside of AWS, as shown in the following diagram:


Your application
Instrument your application with the ADOT SDK
Configure the ADOT collector for data export
ADOT Collector gathers data
Backend
Backend

AWS regularly updates ADOT to add functionality and align with the OpenTelemetry framework. Updates and future plans for developing ADOT are part of a roadmap that is available to the public. ADOT supports several programming languages which include the following:

- Go
- Java
- JavaScript
- Python
- .NET
- Ruby
- PHP

If you are using Python, ADOT can automatically instrument your application. To get started using ADOT, see Introduction and Getting Started with the AWS Distro for OpenTelemetry Collector.

Use the X-Ray SDK

The X-Ray SDK is a set of AWS APIs and libraries that send data to AWS backend services. Use the X-Ray SDK to instrument your application and collect trace data. You cannot use the X-Ray SDK to collect log or metric data.

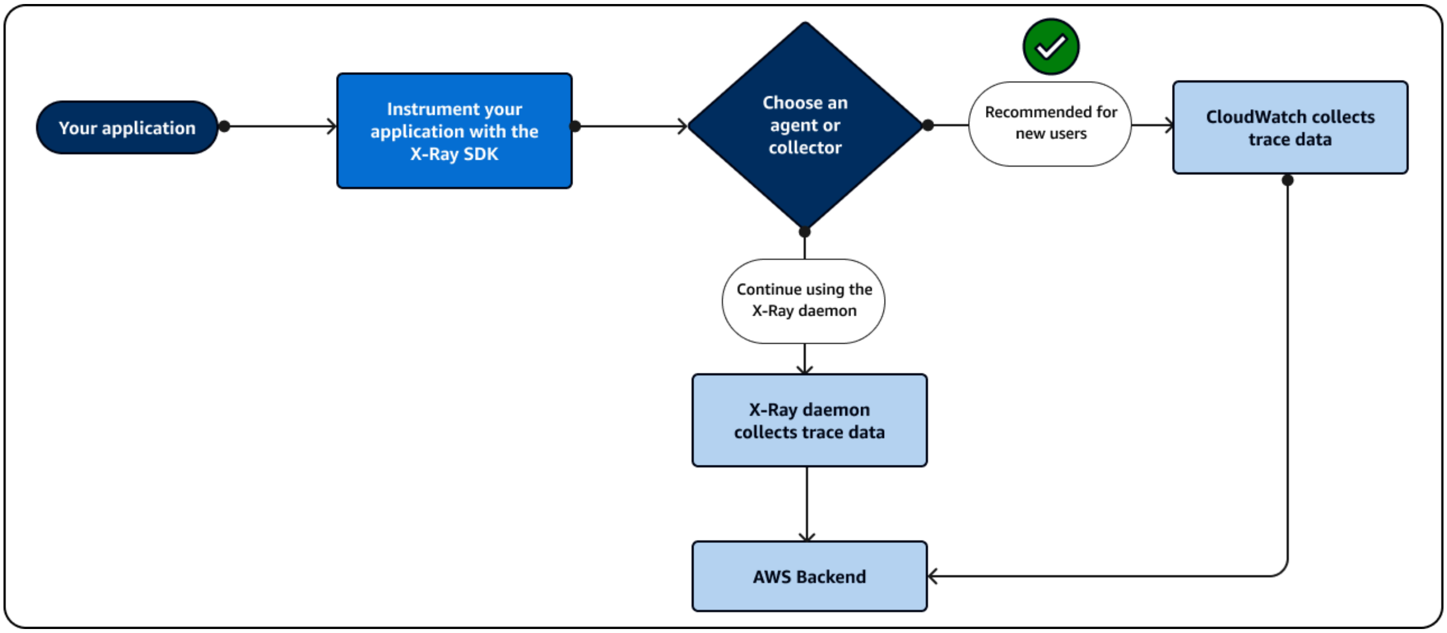
If you are using the X-Ray SDK, you have the following options, in combination with an agent:

- Use the X-Ray SDK with the AWS X-Ray daemon – Use this if you don't want to update your application code.
- Use the X-Ray SDK with the CloudWatch agent – (recommended) The CloudWatch agent is compatible with the X-Ray SDK.

To use the X-Ray SDK, do the following:

- Instrument your application using the X-Ray SDK.
- Configure a collector to tell it where to send data that it collects. You can use either the CloudWatch agent or the X-Ray daemon to collect your trace information.

After the collector or agent receives your data, it sends it to an AWS backend that you specify in the agent configuration. The X-Ray SDK can only send data to an AWS backend as shown in the following diagram:


Your application
Instrument your application with the X-Ray SDK
Choose an agent or collector
✓
Recommended for new users
CloudWatch collects trace data
Continue using the X-Ray daemon
X-Ray daemon collects trace data
AWS Backend

If you are using Java, you can use the X-Ray SDK to automatically instrument your application. To get started using the X-Ray SDK, see the libraries associated with the following programming languages:

- Go
- Java
- Node.js

- Python
- .NET
- Ruby

Use a console

Use a console if you want a graphical user interface (GUI) that requires minimal coding. Users that are new to X-Ray can get started quickly using pre-built visualizations, and performing basic tasks. You can do the following directly from the console:

- Enable X-Ray.
- View high-level summaries of your application's performance.
- Check the health status of your applications.
- Identify high-level errors.
- View basic trace summaries.

You can use either the Amazon CloudWatch console at https://console.aws.amazon.com/cloudwatch/ or the X-Ray console at https://console.aws.amazon.com/xray/home to interact with X-Ray.

Use the Amazon CloudWatch console

The CloudWatch console includes new X-Ray functionality that is redesigned from the X-Ray console to make it easier to use. If you use the CloudWatch console, you can view CloudWatch logs and metrics along with X-Ray trace data. Use the CloudWatch console to view and analyze data including the following:

- X-Ray traces – View, analyze and filter traces associated with your application as it serves a request. Use these traces to find high latencies, debug errors, and optimize your application workflow. View a trace map and service map to see visual representations of your application workflow.
- Logs – View, analyze and filter logs that your application produces. Use logs to troubleshoot errors and set up monitoring based on specific log values.
- Metrics – Measure and monitor your application performance using metrics that your resources emit or create your own metrics. View these metrics in graphs and charts.

- Monitoring networks and infrastructure – Monitor major networks for outages and the health and performance of your infrastructure including containerized applications, other AWS services, and clients.
- All of the functionality from the X-Ray console listed in the following **Use the X-Ray console** section.

For more information about the CloudWatch console, see Getting started with Amazon CloudWatch.

Login the Amazon CloudWatch console at https://console.aws.amazon.com/cloudwatch/.

Use the X-Ray console

The X-Ray console offers distributed tracing for application requests. Use the X-Ray console if you want a simpler console experience or don't want to update your application code. AWS is no longer developing the X-Ray console. The X-Ray console contains the following features for instrumented applications:

- Insights – Automatically detect anomalies in your application's performance and find the underlying causes. Insights are included in the CloudWatch console under **Insights**. For more information, see the **Use X-Ray Insights** in Use the X-Ray console.
- Service map – View a graphical structure of your application and its connections with clients, resources, services, and dependencies.
- Traces – See an overview of traces that are generated by your application as it serves a request. Use trace data to understand how your application performs against basic metrics including HTTP response and response time.
- Analytics – Interpret, explore and analyze trace data using graphs for response time distribution.
- Configuration – Create customized traces to change the default configurations for the following:
  - Sampling – Create a rule that defines how often to sample your application for trace information. For more information, see **Configure sampling rules** in Use the X-Ray console .
  - Encryption – Encrypt data at rest using a key that you can audit or disable using AWS Key Management Service.
  - Groups – Use a filter expression to define a group of traces with a common feature such as the name of a url or a response time. For more information, see Configure groups.

Login the X-Ray console at https://console.aws.amazon.com/xray/home.

Explore the X-Ray console

Use the X-Ray console to view a map of services and associated traces for requests that your applications serve, and to configure groups and sampling rules which affect how traces are sent to X-Ray.

> **Note**
>
> The X-Ray Service map and CloudWatch ServiceLens map have been combined into the X-Ray trace map within the Amazon CloudWatch console. Open the CloudWatch console and choose **Trace Map** under **X-Ray traces** from the left navigation pane.
> CloudWatch now includes Application Signals, which can discover and monitor your application services, clients, Synthetics canaries, and service dependencies. Use Application Signals to see a list or visual map of your services, view health metrics based on your service level objectives (SLOs), and drill down to see correlated X-Ray traces for more detailed troubleshooting.

The primary X-Ray console page is the trace map, which is a visual representation of the JSON service graph that X-Ray generates from the trace data generated by your applications. The map consists of service nodes for each application in your account that serves requests, upstream client nodes that represent the origins of the requests, and downstream service nodes that represent web services and resources used by an application while processing a request. There are additional pages for viewing traces and trace details, and configuring groups and sampling rules.

View the console experience for X-Ray and compare with the CloudWatch console in the following sections.

Explore the X-Ray and CloudWatch consoles

- Using the X-Ray trace map
- Viewing traces and trace details
- Using filter expressions
- Cross-account tracing
- Tracing event-driven applications
- Using latency histograms
- Using X-Ray insights

- Interacting with the Analytics console
- Configuring groups
- Configuring sampling rules
- Console deep linking

Using the X-Ray trace map

View the X-Ray trace map to identify services where errors are occurring, connections with high latency, or traces for requests that were unsuccessful.

> **Note**
>
> CloudWatch now includes Application Signals, which can discover and monitor your application services, clients, synthetics canaries, and service dependencies. Use Application Signals to see a list or visual map of your services, view health metrics based on your service level objectives (SLOs), and drill down to see correlated X-Ray traces for more detailed troubleshooting.
>
> The X-Ray service map and CloudWatch ServiceLens map are combined into the X-Ray trace map within the Amazon CloudWatch console. Open the CloudWatch console and choose **Trace Map** under **X-Ray traces** from the left navigation pane.

Viewing the trace map

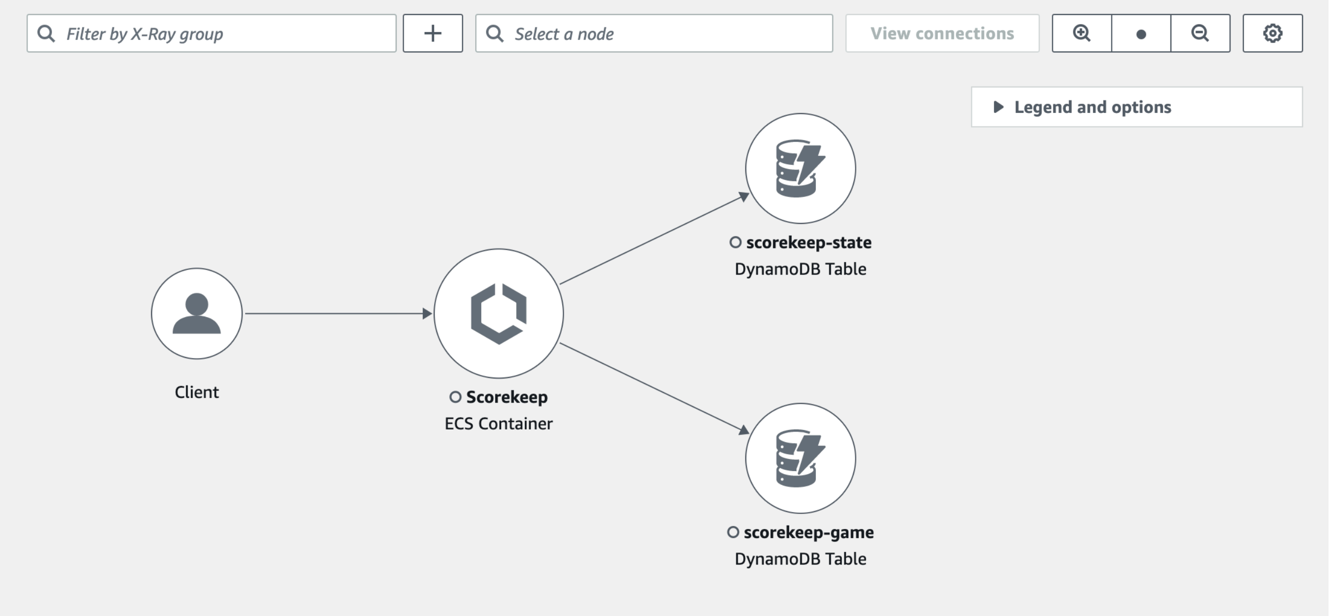
The trace map is a visual representation of the trace data that's generated by your applications. The map shows service nodes that serve requests, upstream client nodes that represent the origins of the requests, and downstream service nodes that represent web services and resources that are used by an application while processing a request.

The trace map displays a connected view of traces across event-driven applications that use Amazon SQS and Lambda. For more information, see tracing event-driven applications. The trace map also supports cross-account tracing, displaying nodes from multiple accounts in a single map.

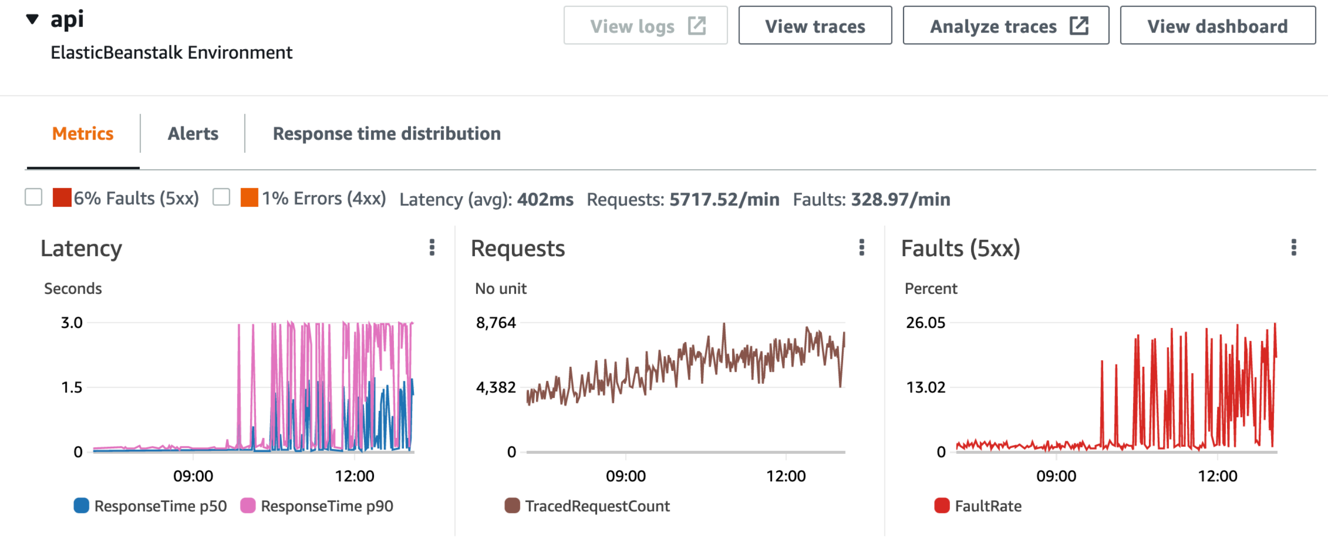
CloudWatch console

To view the trace map in the CloudWatch console

1. Open the CloudWatch console. Choose **Trace Map** under the **X-Ray Traces** section in the left navigation pane.



2. Choose a service node to view requests for that node, or an edge between two nodes to view requests that traveled that connection.
3. Additional information is displayed below the trace map, including tabs for metrics, alerts, and response time distribution. On the **Metrics** tab, select a range within each graph to drill down to view more detail, or choose **Faults** or **Errors** options to filter traces. On the **Response time distribution** tab, select a range within the graph to filter traces by response time.



□ 🟥 6% Faults (5xx) □ 🟧 1% Errors (4xx) Latency (avg): **402ms** Requests: **5717.52/min** Faults: **328.97/min**

4. View traces by choosing **View traces**, or if a filter has been applied, choose **View filtered traces**.

5. Choose **View logs** to see CloudWatch logs associated with the selected node. Not all trace map nodes support viewing logs. See troubleshooting CloudWatch logs for more information.

The trace map indicates issues within each node by outlining it with colors:

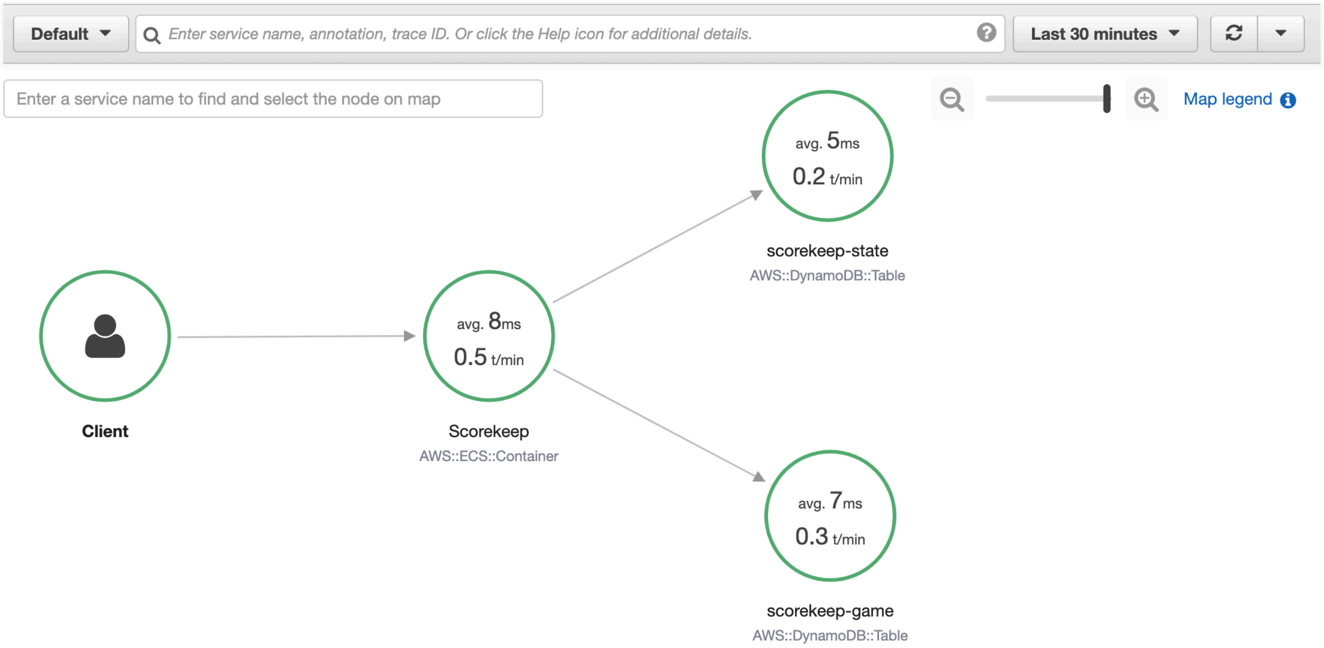
- **Red** for server faults (500 series errors)
- **Yellow** for client errors (400 series errors)
- **Purple** for throttling errors (429 Too Many Requests)

If your trace map is large, use the on-screen controls or mouse to zoom in and out and move the map around.

X-Ray console

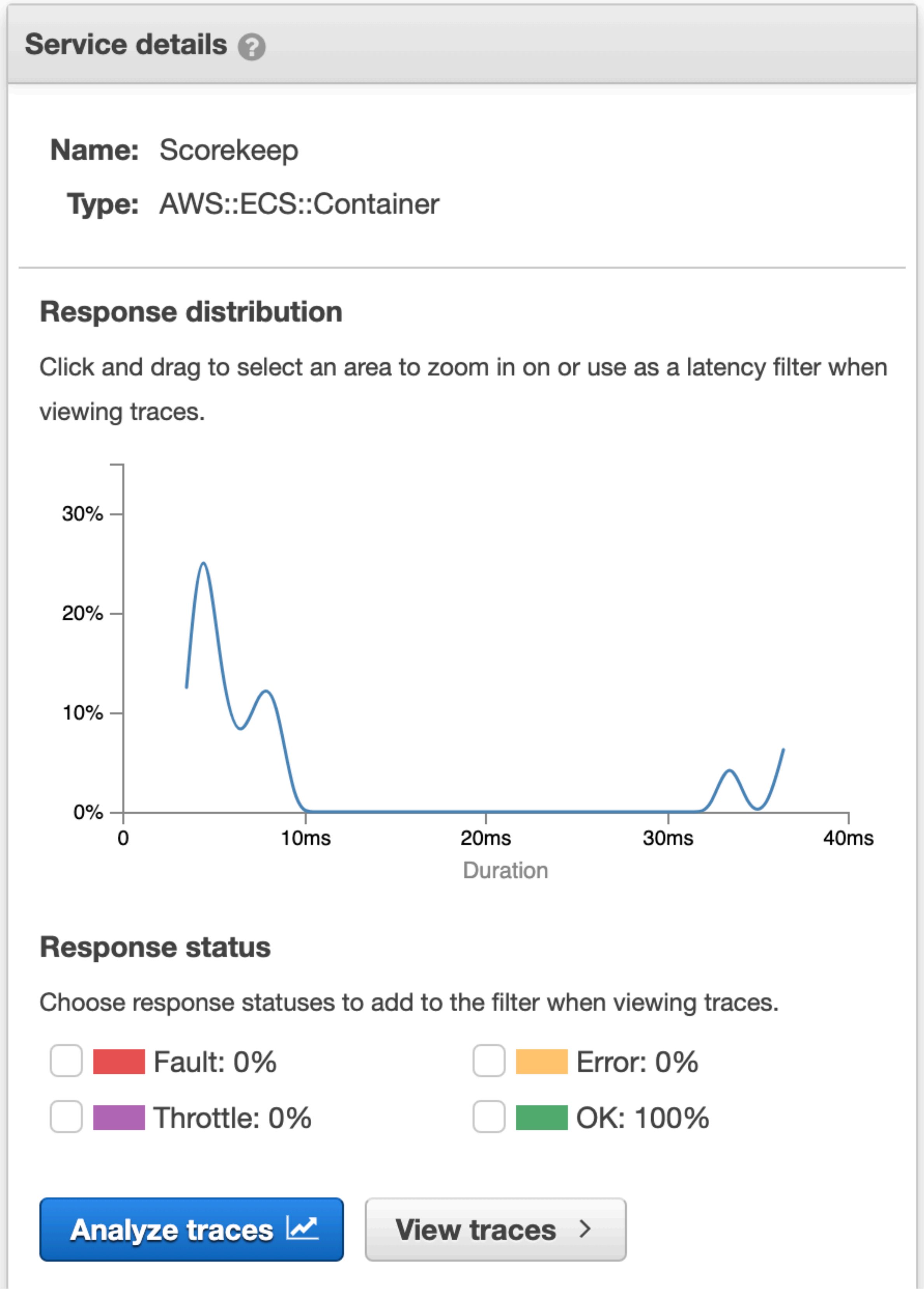
To view the Service map

1. Open the X-Ray console. The service map is displayed by default. You can also choose **Service Map** from the left navigation pane.



2. Choose a service node to view requests for that node, or an edge between two nodes to view requests that traveled that connection.

3. Use the response distribution histogram to filter traces by duration, and select status codes for which you want to view traces. Then choose **View traces** to open the trace list with the filter expression applied.



Service details ?

Name: Scorekeep

Type: AWS::ECS::Container

Response distribution

Click and drag to select an area to zoom in on or use as a latency filter when viewing traces.

Response status

Choose response statuses to add to the filter when viewing traces.

☐ 🟥 Fault: 0% ☐ 🟧 Error: 0%

☐ 🟪 Throttle: 0% ☐ 🟩 OK: 100%

The service map indicates the health of each node by coloring it based on the ratio of successful calls to errors and faults:

- **Green** for successful calls
- **Red** for server faults (500 series errors)
- **Yellow** for client errors (400 series errors)
- **Purple** for throttling errors (429 Too Many Requests)

If your service map is large, use the on-screen controls or mouse to zoom in and out and move the map around.

Note

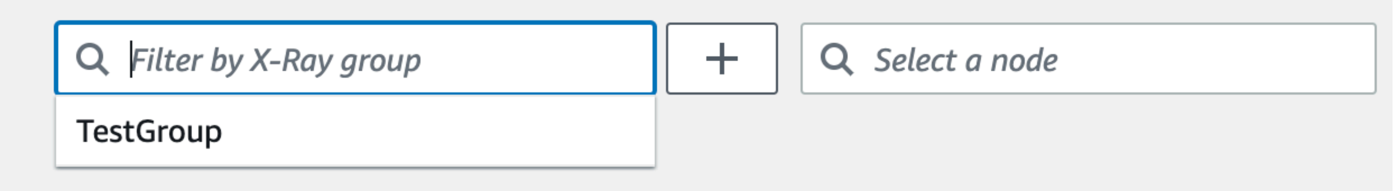
The X-Ray trace map can display up to 10,000 nodes. In rare scenarios where the total number of service nodes exceeds this limit, you may receive an error and be unable to display a complete trace map in the console.

Filtering the trace map by group

Using a filter expression, you can define criteria by which to include traces within a group. Use the following steps to then display that specific group in the trace map.

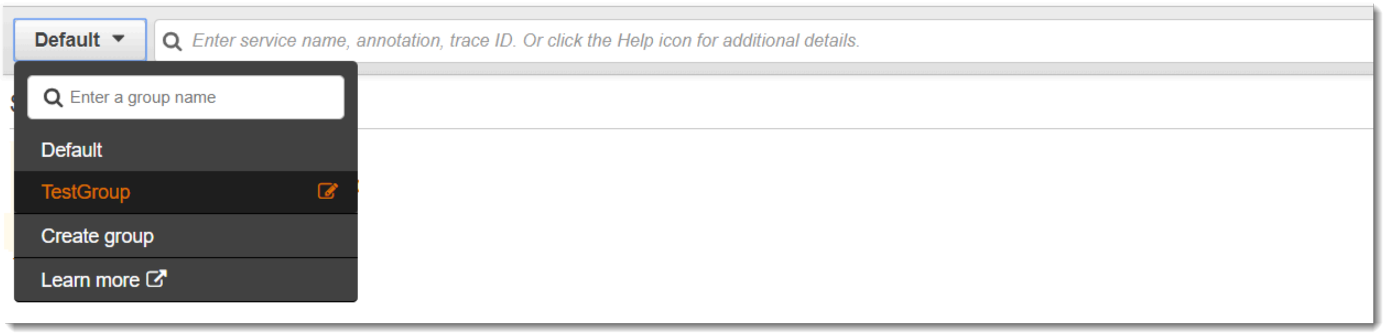
CloudWatch console

Choose a group name from the group filter on the top-left of the trace map.



X-Ray console

Choose a group name from the drop-down menu to the left of the search bar.


Default
Enter service name, annotation, trace ID. Or click the Help icon for additional details.
Enter a group name
Default
TestGroup
Create group
Learn more

The service map will now be filtered to display traces that match the filter expression of the selected group.

Trace map legend and options

The trace map includes a legend and several options for customizing the map display.

CloudWatch console

Choose the **Legend and options** drop-down at the top-right of the map. Choose what is displayed within nodes, including:

- **Metrics** displays the average response time and number of traces sent per minute during the chosen time range.
- **Nodes** displays the service icon within each node.

Choose additional map settings from the **Preferences** pane, which can be accessed via the gear icon at the top-right of the map. These settings include selecting which metric is used to determine the size of each node, and which canaries should be displayed on the map.

X-Ray console

Display the service map legend by choosing the **Map legend** link at the top-right of the map. Service map options can be chosen at the bottom-right of the trace map, including:

- **Service Icons** toggles what is displayed within each node, displaying either the service icon, or the average response time and number of traces sent per minute during the chosen time range.
- **Node sizing: None** sets all nodes to the same size.

- **Node sizing: Health** sizes nodes by the number of impacted requests including errors, faults, or throttled requests.
- **Node sizing: Traffic** sizes nodes by the total number of requests.

Viewing traces and trace details

Use the **Traces** page in the X-Ray console to find traces by URL, response code, or other data from the trace summary. After selecting a trace from the trace list, the **Trace details** page displays a map of service nodes that are associated with the selected trace and a timeline of trace segments.

Viewing traces

CloudWatch console

To view traces in the CloudWatch console

1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. In the left navigation pane, choose **X-Ray traces**, then choose **Traces**. You can filter by group or enter a filter expression. This filters the traces that are displayed in the **Traces** section at the bottom of the page.

   Alternatively, you can use the service map to navigate to a specific service node, and then view traces. This opens the **Traces** page with a query already applied.
3. Refine your query in the **Query refiners** section. To filter traces by a common attribute, choose an option from the down arrow next to **Refine query by**. The options include the following:

   - Node – Filter traces by service node.
   - Resource ARN – Filter traces by a resource associated with a trace. Examples of these resources include Amazon Elastic Compute Cloud (Amazon EC2) instance, an AWS Lambda function, or an Amazon DynamoDB table.
   - User – Filter traces with a user ID.
   - Error root cause message – Filter traces by error root cause.
   - URL – Filter traces by a URL path used by your application.
   - HTTP status code – Filter traces by the HTTP status code returned by your application. You can specify a custom response code or select from the following:

  - `200` – The request was successful.
  - `401` – The request lacked valid authentication credentials.
  - `403` – The request lacked valid permissions.
  - `404` – The server could not find the requested resource.
  - `500` – The server encountered an unexpected condition and generated an internal error.

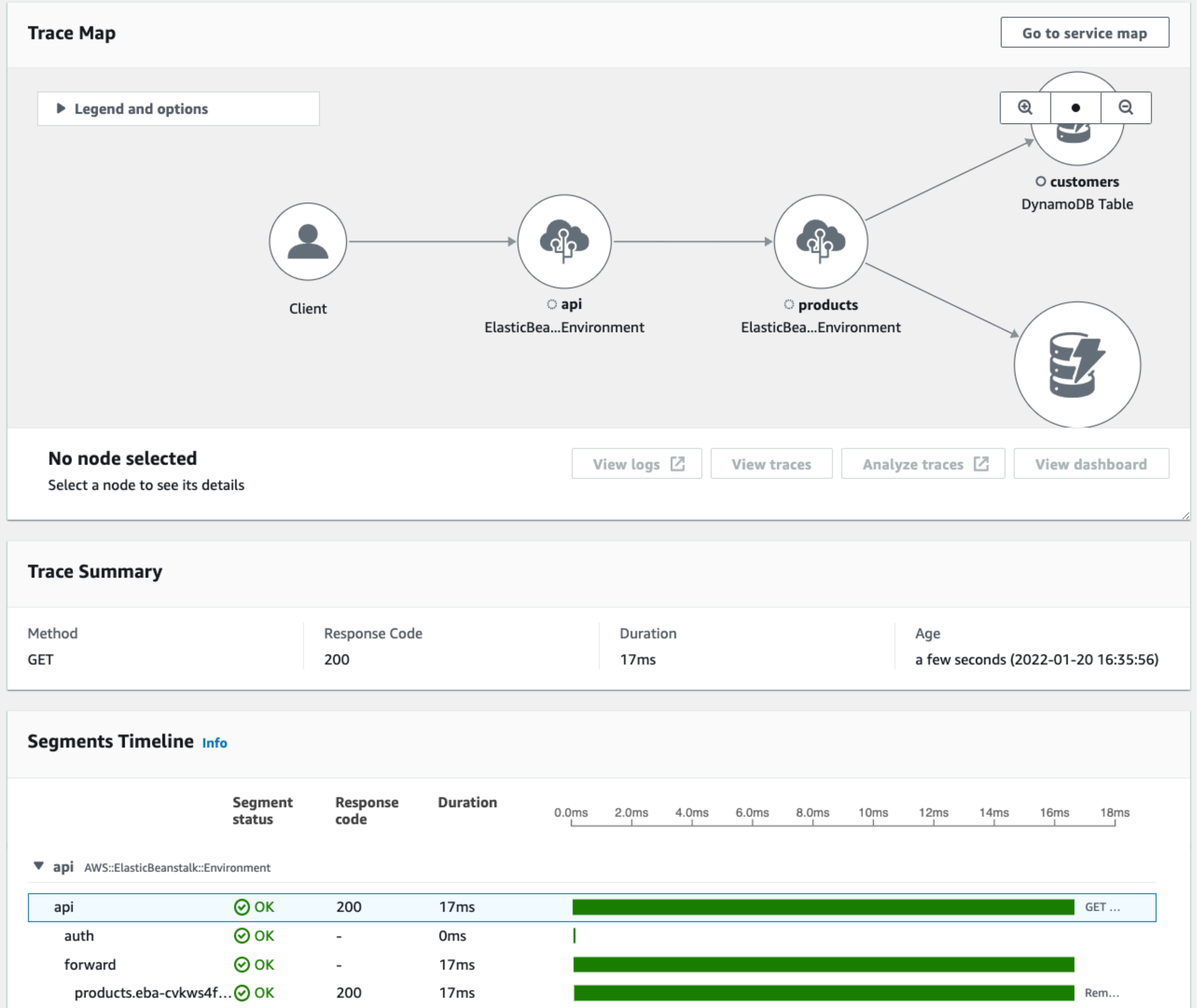
   Choose one or more entries and then choose **Add to query** to add to the filter expression at the top of the page.

4. To find a single trace, enter a trace ID directly into the query field. You can use X-Ray format or World Wide Web Consortium (W3C) format. For example, a trace that's created using the AWS Distro for OpenTelemetry is in W3C format.

   > **Note**
   >
   > When you query traces that are created with a W3C-format trace ID, the console displays the matching trace in X-Ray format. For example, if you query for `4efaaf4d1e8720b39541901950019ee5` in W3C format, the console displays the X-Ray equivalent: `1-4efaaf4d-1e8720b39541901950019ee5`.

5. Choose **Run query** at any time to display a list of matching traces within the **Traces** section at the bottom of the page.
6. To display the **Trace details** page for a single trace, select a trace ID from the list.

   The following image shows a **Trace map** containing service nodes associated with the trace and edges between the nodes representing the path taken by segments that compose the trace. A **Trace summary** follows the **Trace Map**. The summary contains information about a sample GET operation, its **Response Code**, the **Duration** that the trace took to run, and the **Age** of the request. The **Segments Timeline** follows the **Trace Summary** that shows the duration of trace segments and subsegments.



No node selected

Select a node to see its details

Method	Response Code	Duration	Age
GET	200	17ms	a few seconds (2022-01-20 16:35:56)

▼ api AWS::ElasticBeanstalk::Environment

	Segment status	Response code	Duration
api	✓ OK	200	17ms
auth	✓ OK	-	0ms
forward	✓ OK	-	17ms
products.eba-cvkws4f...	✓ OK	200	17ms

If you have an event-driven application that uses Amazon SQS and Lambda, you can see a connected view of traces for each request in the **Trace map**. In the map, traces from message producers are linked to traces from AWS Lambda consumers and are displayed as a dashed-line edge. For more information about event-driven applications, see Tracing event-driven applications.

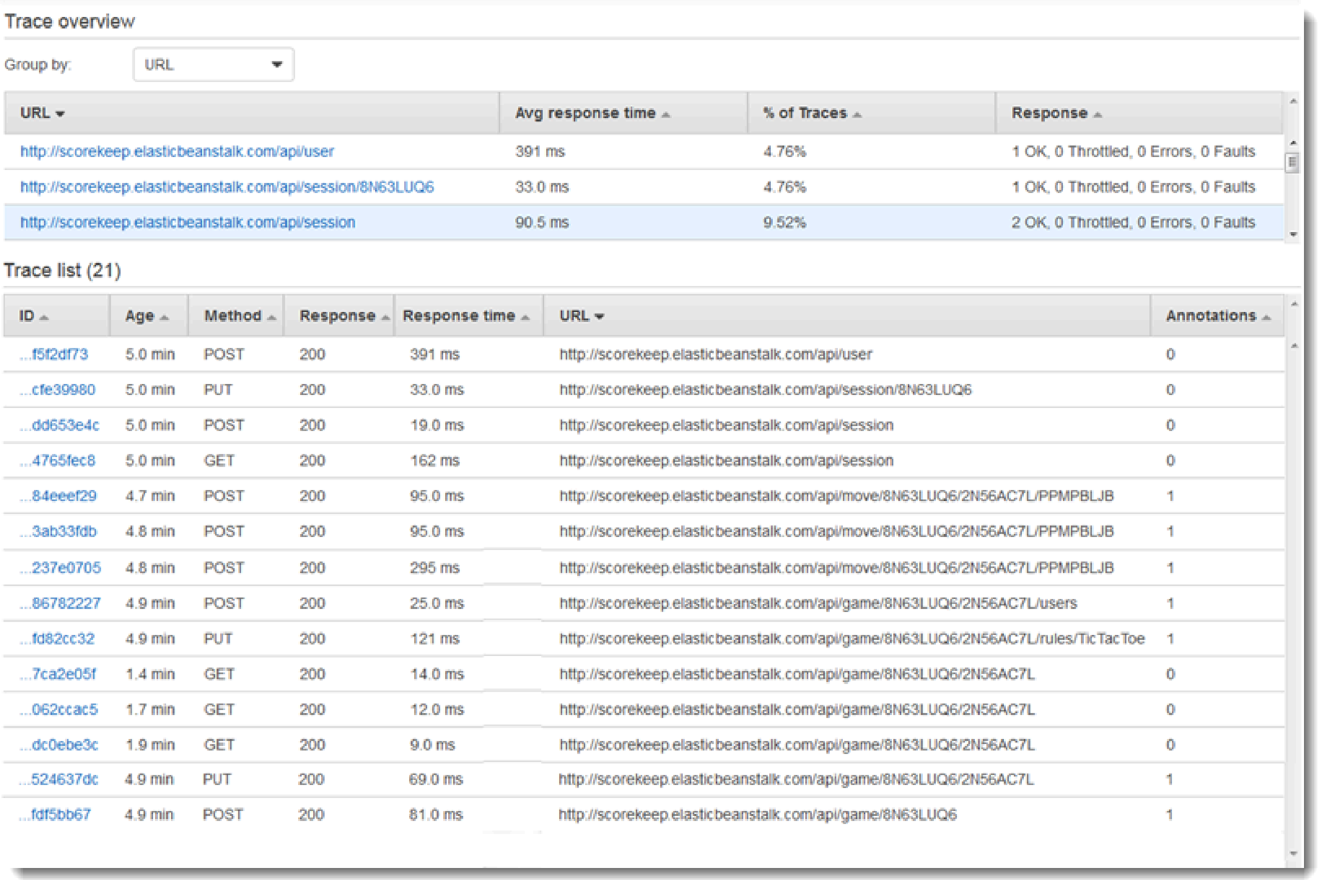
The **Traces** and **Trace details** pages also support cross-account tracing, which can list traces from multiple accounts in the trace list and inside a single trace map.

X-Ray console

To view traces in the X-Ray console

1. Open the Traces page in the X-Ray console. The **Trace overview** panel shows a list of traces that are grouped by common features including **Error root causes**, **ResourceARN**, and **InstanceId**.

2. To select a common feature to view a grouped set of traces, expand the down arrow next to **Group by**. The following illustration shows a trace overview of traces that are grouped by URL for the AWS X-Ray sample application, and a list of associated traces.



Trace overview

Group by: URL

URL	Avg response time	% of Traces	Response
http://scorekeep.elasticbeanstalk.com/api/user	391 ms	4.76%	1 OK, 0 Throttled, 0 Errors, 0 Faults
http://scorekeep.elasticbeanstalk.com/api/session/8N63LUQ6	33.0 ms	4.76%	1 OK, 0 Throttled, 0 Errors, 0 Faults
http://scorekeep.elasticbeanstalk.com/api/session	90.5 ms	9.52%	2 OK, 0 Throttled, 0 Errors, 0 Faults

Trace list (21)

ID	Age	Method	Response	Response time	URL	Annotations
...f5f2df73	5.0 min	POST	200	391 ms	http://scorekeep.elasticbeanstalk.com/api/user	0
...cfe39980	5.0 min	PUT	200	33.0 ms	http://scorekeep.elasticbeanstalk.com/api/session/8N63LUQ6	0
...dd653e4c	5.0 min	POST	200	19.0 ms	http://scorekeep.elasticbeanstalk.com/api/session	0
...4765fec8	5.0 min	GET	200	162 ms	http://scorekeep.elasticbeanstalk.com/api/session	0
...84eeef29	4.7 min	POST	200	95.0 ms	http://scorekeep.elasticbeanstalk.com/api/move/8N63LUQ6/2N56AC7L/PPMPBLJB	1
...3ab33fdb	4.8 min	POST	200	95.0 ms	http://scorekeep.elasticbeanstalk.com/api/move/8N63LUQ6/2N56AC7L/PPMPBLJB	1
...237e0705	4.8 min	POST	200	295 ms	http://scorekeep.elasticbeanstalk.com/api/move/8N63LUQ6/2N56AC7L/PPMPBLJB	1
...86782227	4.9 min	POST	200	25.0 ms	http://scorekeep.elasticbeanstalk.com/api/game/8N63LUQ6/2N56AC7L/users	1
...fd82cc32	4.9 min	PUT	200	121 ms	http://scorekeep.elasticbeanstalk.com/api/game/8N63LUQ6/2N56AC7L/rules/TicTacToe	1
...7ca2e05f	1.4 min	GET	200	14.0 ms	http://scorekeep.elasticbeanstalk.com/api/game/8N63LUQ6/2N56AC7L	0
...062ccac5	1.7 min	GET	200	12.0 ms	http://scorekeep.elasticbeanstalk.com/api/game/8N63LUQ6/2N56AC7L	0
...dc0ebe3c	1.9 min	GET	200	9.0 ms	http://scorekeep.elasticbeanstalk.com/api/game/8N63LUQ6/2N56AC7L	0
...524637dc	4.9 min	PUT	200	69.0 ms	http://scorekeep.elasticbeanstalk.com/api/game/8N63LUQ6/2N56AC7L	1
...fdf5bb67	4.9 min	POST	200	81.0 ms	http://scorekeep.elasticbeanstalk.com/api/game/8N63LUQ6	1

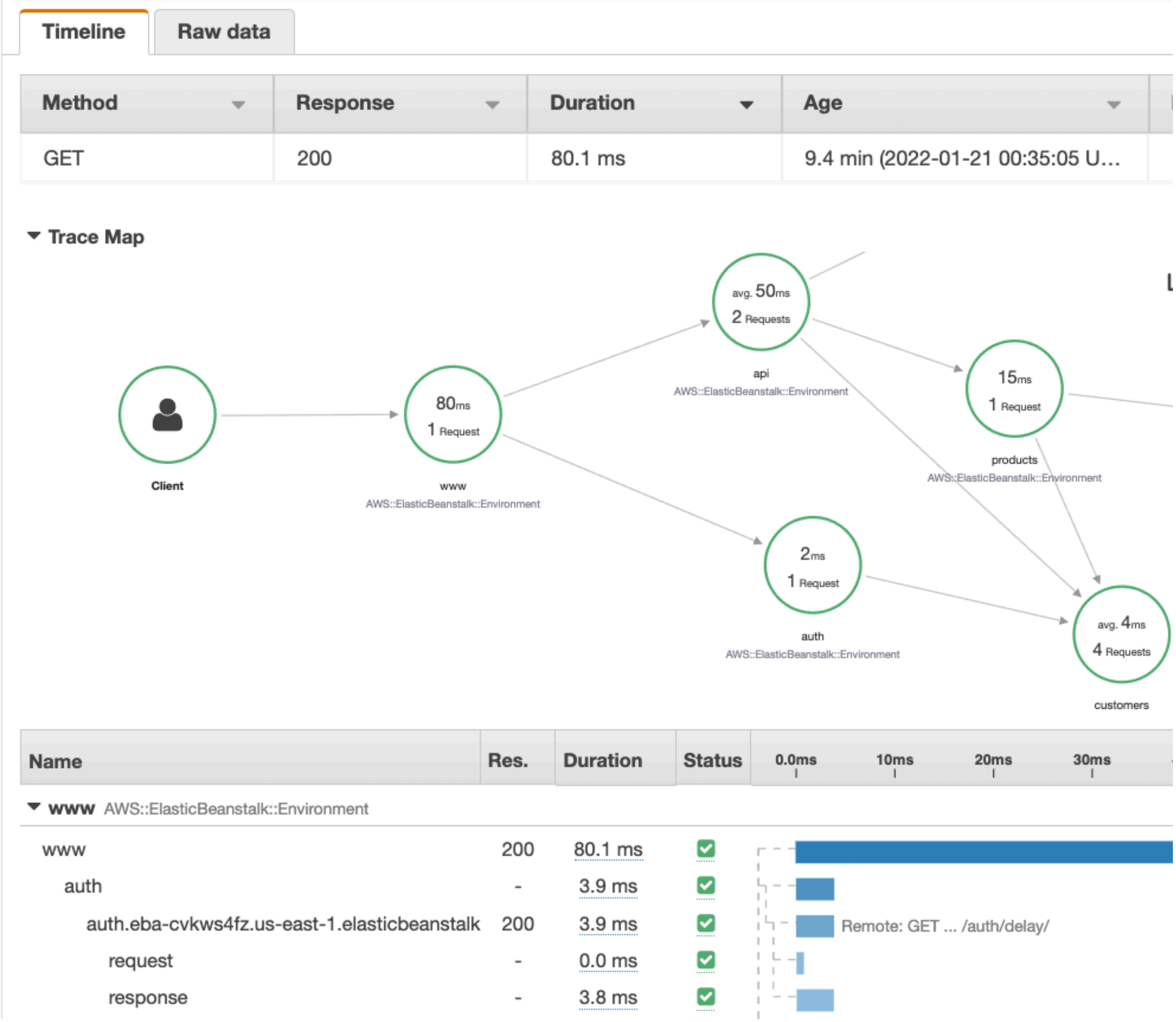
3. Choose the **ID** of a trace to view it under the **Trace list**. You can also choose **Service map** in the navigation pane to view traces for a specific service node. Then you can view traces that are associated with that node.

   The **Timeline** tab shows the request flow for the trace, and includes the following:

   - A map of the path for each segment in the trace.
   - How long it took for the segment to reach a node in the trace map.

- How many requests were made to the node in the trace map.

The following illustration shows an example **Trace Map** associated with a GET request made to a sample application. The arrows show the path that each segment took to complete the request. The service nodes show the number of requests made during the GET request.


Timeline
Raw data
Trace Map
avg. 50ms
2 Requests
api
AWS::ElasticBeanstalk::Environment
15ms
1 Request
products
AWS::ElasticBeanstalk::Environment
Client
80ms
1 Request
www
AWS::ElasticBeanstalk::Environment
2ms
1 Request
auth
AWS::ElasticBeanstalk::Environment
avg. 4ms
4 Requests
customers

Method	Response	Duration	Age
GET	200	80.1 ms	9.4 min (2022-01-21 00:35:05 U...

www AWS::ElasticBeanstalk::Environment

Name	Res.	Duration	Status	0.0ms 10ms 20ms 30ms
www	200	80.1 ms	☑	
auth	-	3.9 ms	☑	
auth.eba-cvkws4fz.us-east-1.elasticbeanstalk	200	3.9 ms	☑	Remote: GET ... /auth/delay/
request	-	0.0 ms	☑	
response	-	3.8 ms	☑	

For more information about the **Timeline** tab, see the following **Exploring the trace timeline** section.

The **Raw data** tab shows information about the trace, and the segments and subsegments that compose the trace, in JSON format. This information may include the following:

- Timestamps
- Unique IDs
- Resources associated with the segment or subsegment
- The source, or origin, of the segment or subsegment
- Additional information about the request to your application such as the response from an HTTP request

Exploring the trace timeline

The **Timeline** section shows a hierarchy of segments and subsegments next to a horizontal bar that corresponds to time they used to complete their tasks. The first entry in the list is the segment, which represents all data recorded by the service for a single request. Subsegments are indented and listed following the segment. Columns contain information about each segment.

CloudWatch console

In the CloudWatch console, the **Segments Timeline** provides the following information:

- The first column: Lists the segments and subsegments in the selected trace.
- The **Segment status** column: Lists the status outcome of each segment and subsegment.
- The **Response code** column: Lists an HTTP response status code to a browser request made by the segment or subsegment, when available.
- The **Duration** column: Lists how long the segment or subsegment ran.
- The **Hosted in** column: Lists the namespace or environment where the segment or subsegment is ran, if applicable. For more information, see Dimensions collected and dimension combinations.
- The last column: Displays horizontal bars that correspond to the duration that the segment or subsegment ran, in relation to the other segments or subsegments in the timeline.

To group the list of segments and subsegments by service node, turn on **Group by nodes**.

X-Ray console

In the trace details page, choose the **Timeline** tab to see the timeline for each segment and subsegment that makes up a trace.

In the X-Ray console, the **Timeline** provides the following information:

- The **Name** column: Lists the names of the segments and subsegments in the trace.
- The **Res.** column: Lists an HTTP response status code to a browser request made by the segment or subsegment, when available.
- The **Duration** column: Lists how long the segment or subsegment ran.
- The **Status** column: Lists the outcome of the segment or subsegment status.
- The last column: Displays horizontal bars that correspond to the duration that the segment or subsegment ran, in relation to the other segments or subsegments in the timeline.

To see the raw trace data that the console uses to generate the timeline, choose the **Raw data** tab. The raw data shows you information about the trace, and the segments and subsegments that compose the trace in `JSON` format. This information may include the following:

- Timestamps
- Unique IDs
- Resources associated with the segment or subsegment
- The source, or origin, of the segment or subsegment
- Additional information about the request to your application such as the response from an HTTP request.

When you use an instrumented AWS SDK, HTTP, or SQL client to make calls to external resources, the X-Ray SDK records subsegments automatically. You can also use the X-Ray SDK to record custom subsegments for any function or block of code. Additional subsegments that are recorded while a custom subsegment are open become children of the custom subsegment.

Viewing segment details

From the trace **Timeline**, choose the name of a segment to view its details.

The **Segment details** panel shows the **Overview**, **Resources**, **Annotations**, **Metadata**, **Exceptions**, and **SQL** tabs. The following apply:

- The **Overview** tab shows information about the request and response. Information includes the name, start time, end time, duration, the request URL, request operation, request response code, and any errors and faults.

- The **Resources** tab for a segment shows information from the X-Ray SDK and about the AWS resources running your application. Use the Amazon EC2, AWS Elastic Beanstalk, or Amazon ECS plugins for the X-Ray SDK to record service-specific resource information. For more information about plugins, see the **Service plugins** section in Configuring the X-Ray SDK for Java.
- The remaining tabs show **Annotations**, **Metadata**, and **Exceptions** that are recorded for the segment. Exceptions are captured automatically when they are generated from an instrumented request. Annotations and metadata contain additional information that you record by using the operations that the X-Ray SDK provides. To add annotations or metadata to your segments, use the X-Ray SDK. For more information, see the language-specific link listed under Instrumenting your application with AWS X-Ray SDKs in Instrumenting your application for AWS X-Ray.

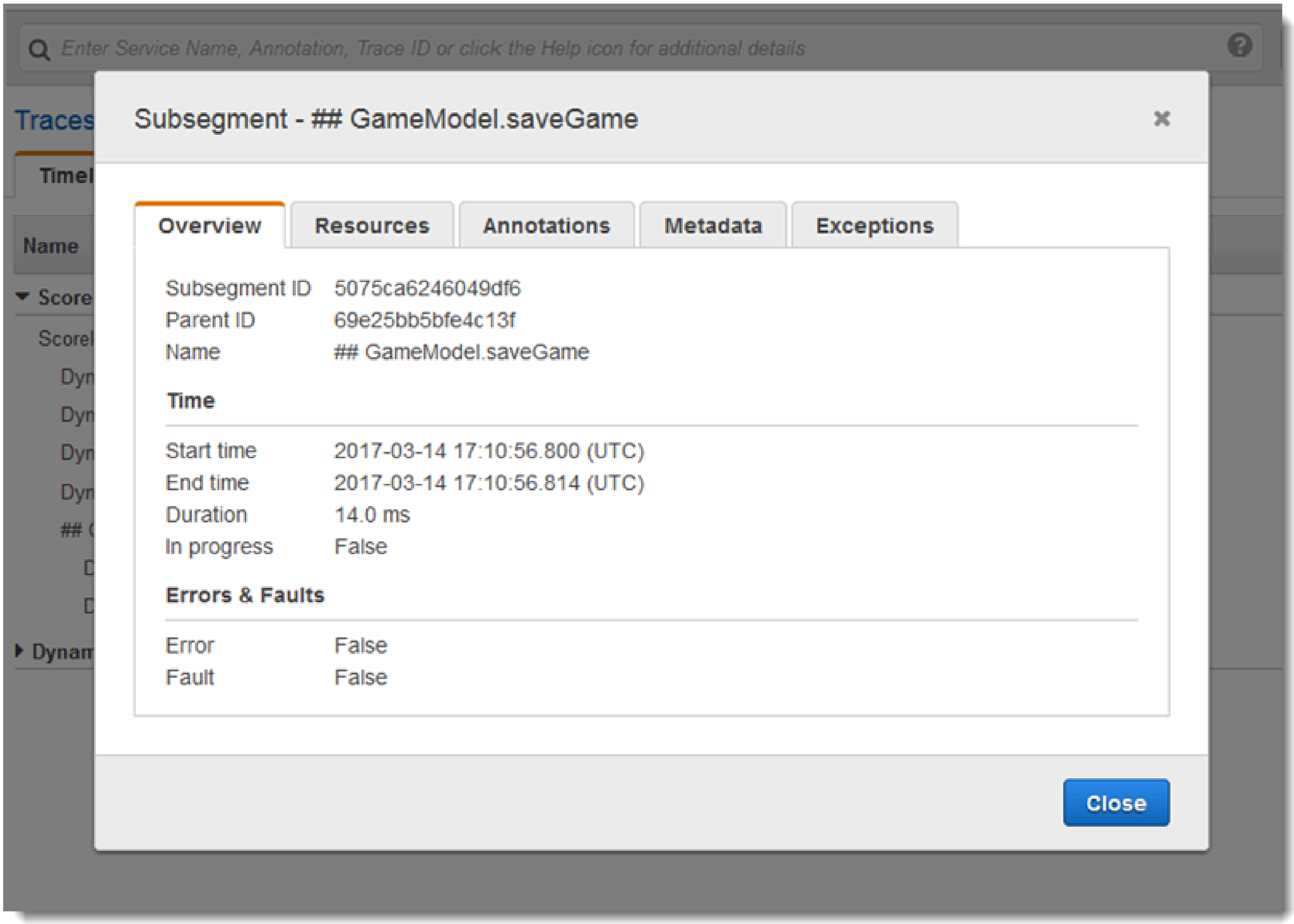
Viewing subsegment details

From the trace timeline, choose the name of a subsegment to view its details:

- The **Overview** tab contains information about the request and response. This includes the name, start time, end time, duration, the request URL, request operation, request response code, and any errors and faults. For subsegments generated with instrumented clients, the **Overview** tab contains information about the request and response from your application's point of view.
- The **Resources** tab for a subsegment shows details about the AWS resources that were used to run the subsegment. For example, the resources tab may include an AWS Lambda function ARN, information about a DynamoDB table, any operation that is called, and request ID.
- The remaining tabs show **Annotations**, **Metadata**, and **Exceptions** recorded on the subsegment. Exceptions are captured automatically when they are generated from an instrumented request. Annotations and metadata contain additional information that you record by using the operations that the X-Ray SDK provides. Use the X-Ray SDK to add annotations or metadata to your segments. For more information, see the language-specific link listed under **Instrumenting your application with AWS X-Ray SDKs** in Instrumenting your application for AWS X-Ray.

For custom subsegments, the **Overview** tab shows the name of the subsegment, which you can set to specify the area of the code or function that it records. For more information, see the language-specific link listed under **Instrumenting your application with AWS X-Ray SDKs** in Generating custom subsegments with the X-Ray SDK for Java.

The following image shows the **Overview** tab for a custom subsegment. The overview contains the subsegment ID, parent ID, Name, start and end times, duration, status and errors or faults.



Subsegment - ## GameModel.saveGame

Subsegment ID	5075ca6246049df6
Parent ID	69e25bb5bfe4c13f
Name	## GameModel.saveGame

Time

Start time	2017-03-14 17:10:56.800 (UTC)
End time	2017-03-14 17:10:56.814 (UTC)
Duration	14.0 ms
In progress	False

Errors & Faults

Error	False
Fault	False

The **Metadata** tab for a custom subsegment contains information in JSON format about resources used by that subsegment.

Using filter expressions

Use *filter expressions* to view a trace map or traces for a specific request, service, connection between two services (an edge), or requests that satisfy a condition. X-Ray provides a filter expression language for filtering requests, services, and edges based on data in request headers, response status, and indexed fields on the original segments.

When you choose a time period of traces to view in the X-Ray console, you might get more results than the console can display. In the upper-right corner, the console shows the number of traces that it scanned and whether there are more traces available. You can use a filter expression to narrow the results to just the traces that you want to find.

Topics

- Filter expression details
- Using filter expressions with groups
- Filter expression syntax
- Boolean keywords
- Number keywords
- String keywords
- Complex keywords
- id function

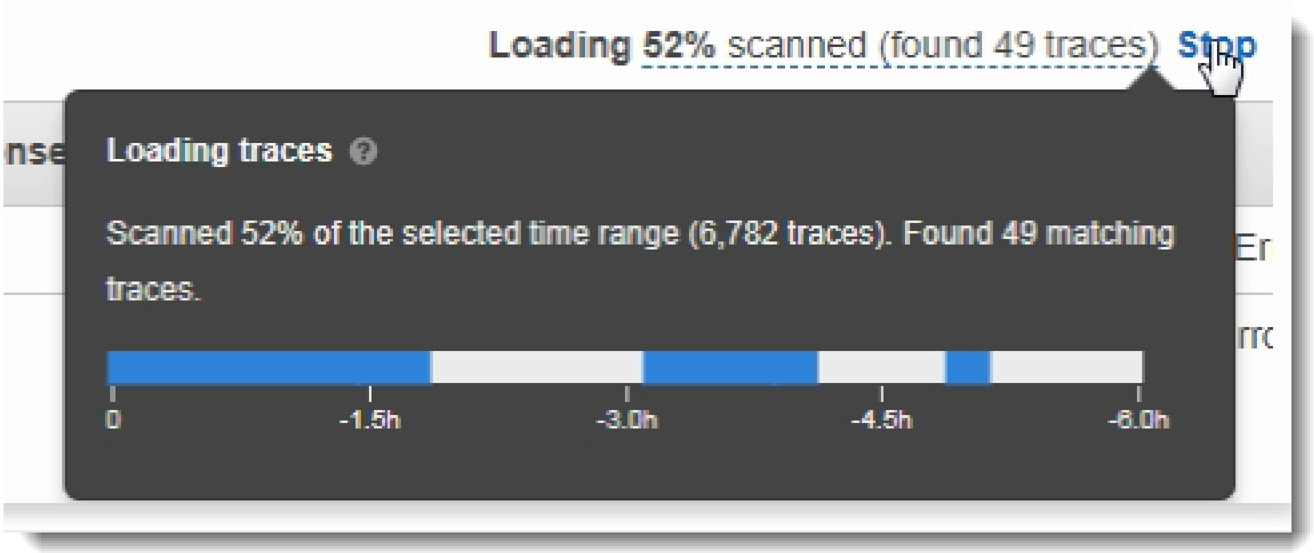
Filter expression details

When you choose a node in the trace map, the console constructs a filter expression based on the service name of the node, and the types of error present based on your selection. To find traces that show performance issues or that relate to specific requests, you can adjust the expression that the console provides or create your own. If you add annotations with the X-Ray SDK, you can also filter based on the presence of an annotation key or the value of a key.

> **Note**
>
> If you choose a relative time range in the trace map and choose a node, the console converts the time range to an absolute start and end time. To ensure that the traces for the node appear in the search results, and avoid scanning times when the node wasn't active, the time range only includes times when the node sent traces. To search relative to the current time, you can switch back to a relative time range in the traces page and scan again.

If there are still more results available than the console can show, the console shows you how many traces matched and the number of traces scanned. The percentage shown is the percentage of the selected time frame that was scanned. To ensure that you see all matching traces represented in the results, narrow your filter expression further, or choose a shorter time frame.

To get the freshest results first, the console starts scanning at the end of the time range and works backward. If there are a large number of traces, but few results, the console splits the time range into chunks and scans them in parallel. The progress bar shows the parts of the time range that have been scanned.


Loading 52% scanned (found 49 traces) Stop

Loading traces

Scanned 52% of the selected time range (6,782 traces). Found 49 matching traces.

Using filter expressions with groups

Groups are a collection of traces that are defined by a filter expression. You can use groups to generate additional service graphs and supply Amazon CloudWatch metrics.

Groups are identified by their name or an Amazon Resource Name (ARN), and contain a filter expression. The service compares incoming traces to the expression and stores them accordingly.

You can create and modify groups by using the dropdown menu to the left of the filter expression search bar.

> **Note**
>
> If the service encounters an error in qualifying a group, that group is no longer included in processing incoming traces and an error metric is recorded.

For more information about groups, see Configuring groups.

Filter expression syntax

Filter expressions can contain a *keyword*, a unary or binary *operator*, and a *value* for comparison.

```
keyword operator value
```

Different operators are available for different types of keyword. For example, `responsetime` is a number keyword and can be compared with operators related to numbers.

Example – requests where response time was greater than 5 seconds

```
responsetime > 5
```

You can combine multiple expressions in a compound expression by using the `AND` or `OR` operators.

Example – requests where the total duration was 5–8 seconds

```
duration >= 5 AND duration <= 8
```

Simple keywords and operators find issues only at the trace level. If an error occurs downstream, but is handled by your application and not returned to the user, a search for `error` will not find it.

To find traces with downstream issues, you can use the complex keywords `service()` and `edge()`. These keywords let you apply a filter expression to all downstream nodes, a single downstream node, or an edge between two nodes. For more granularity, you can filter services and edges by type with the id() function.

Boolean keywords

Boolean keyword values are either true or false. Use these keywords to find traces that resulted in errors.

Boolean keywords

- `ok` – Response status code was 2XX Success.
- `error` – Response status code was 4XX Client Error.
- `throttle` – Response status code was 429 Too Many Requests.
- `fault` – Response status code was 5XX Server Error.
- `partial` – Request has incomplete segments.
- `inferred` – Request has inferred segments.
- `first` – Element is the first of an enumerated list.
- `last` – Element is the last of an enumerated list.
- `remote` – Root cause entity is remote.
- `root` – Service is the entry point or root segment of a trace.

Boolean operators find segments where the specified key is `true` or `false`.

Boolean operators

- none – The expression is true if the keyword is true.
- `!` – The expression is true if the keyword is false.
- `=`,`!=` – Compare the value of the keyword to the string `true` or `false`. These operators act the same as the other operators but are more explicit.

Example – response status is 2XX OK

```
ok
```

Example – response status is not 2XX OK

```
!ok
```

Example – response status is not 2XX OK

```
ok = false
```

Example – last enumerated fault trace has error name "deserialize"

```
rootcause.fault.entity { last and name = "deserialize" }
```

Example – requests with remote segments where coverage is greater than 0.7 and the service name is "traces"

```
rootcause.responsetime.entity { remote and coverage > 0.7 and name = "traces" }
```

Example – requests with inferred segments where the service type is "AWS:DynamoDB"

```
rootcause.fault.service { inferred and name = traces and type = "AWS::DynamoDB" }
```

Example – requests that have a segment with the name "data-plane" as the root

```
service("data-plane") {root = true and fault = true}
```

Number keywords

Use number keywords to search for requests with a specific response time, duration, or response status.

Number keywords

- `responsetime` – Time that the server took to send a response.
- `duration` – Total request duration, including all downstream calls.
- `http.status` – Response status code.
- `index` – Position of an element in an enumerated list.
- `coverage` – Decimal percentage of entity response time over root segment response time. Applicable only for response time root cause entities.

Number operators

Number keywords use standard equality and comparison operators.

- =,!= – The keyword is equal to or not equal to a number value.
- <,<=, >,>= – The keyword is less than or greater than a number value.

Example – response status is not 200 OK

```
http.status != 200
```

Example – request where the total duration was 5–8 seconds

```
duration >= 5 AND duration <= 8
```

Example – requests that completed successfully in less than 3 seconds, including all downstream calls

```
ok !partial duration <3
```

Example – enumerated list entity that has an index greater than 5

```
rootcause.fault.service { index > 5 }
```

Example – requests where the last entity that has coverage greater than 0.8

```
rootcause.responsetime.entity { last and coverage > 0.8 }
```

String keywords

Use string keywords to find traces with specific text in the request headers, or specific user IDs.

String keywords

- `http.url` – Request URL.
- `http.method` – Request method.
- `http.useragent` – Request user agent string.
- `http.clientip` – Requestor's IP address.
- `user` – Value of the user field on any segment in the trace.
- `name` – The name of a service or exception.
- `type` – Service type.
- `message` – Exception message.
- `availabilityzone` – Value of the availabilityzone field on any segment in the trace.
- `instance.id` – Value of the instance ID field on any segment in the trace.
- `resource.arn` – Value of the resource ARN field on any segment in the trace.

String operators find values that are equal to or contain specific text. Values must always be specified in quotation marks.

String operators

- `=`, `!=` – The keyword is equal to or not equal to a number value.
- `CONTAINS` – The keyword contains a specific string.
- `BEGINSWITH` , `ENDSWITH` – The keyword begins or ends with a specific string.

Example – http.url filter

```
http.url CONTAINS "/api/game/"
```

To test if a field exists on a trace, regardless of its value, check to see if it contains the empty string.

Example – user filter

Find all traces with user IDs.

```
user CONTAINS ""
```

Example – select traces with a fault root cause that includes a service named "Auth"

```
rootcause.fault.service { name = "Auth" }
```

Example – select traces with a response time root cause whose last service has a type of DynamoDB

```
rootcause.responsetime.service { last and type = "AWS::DynamoDB" }
```

Example – select traces with a fault root cause whose last exception has the message "access denied for account_id: 1234567890"

```
rootcause.fault.exception { last and message = "Access Denied for account_id:
 1234567890"
```

Complex keywords

Use complex keywords to find requests based on service name, edge name, or annotation value. For services and edges, you can specify an additional filter expression that applies to the service or edge. For annotations, you can filter on the value of an annotation with a specific key using Boolean, number, or string operators.

Complex keywords

- `annotation[`*`key`*`]` – Value of an annotation with field *`key`*. The value of an annotation can be a Boolean, number, or string, so you can use any of the comparison operators of those types. You can use this keyword in combination with the `service` or `edge` keyword. An annotation key that contains dots (periods) must be wrapped in square brackets (**[ ]**).
- `edge(`*`source`*`, `*`destination`*`) {`*`filter`*`}` – Connection between services *`source`* and *`destination`*. Optional curly braces can contain a filter expression that applies to segments on this connection.

- `group.`*`name`* `/ group.`*`arn`* – The value of a group's filter expression, referenced by group name or group ARN.
- `json` – JSON root cause object. See [Getting data from AWS X-Ray]() for steps to create JSON entities programmatically.
- `service(`*`name`*`) {`*`filter`*`}` – Service with name *`name`*. Optional curly braces can contain a filter expression that applies to segments created by the service.

Use the service keyword to find traces for requests that hit a certain node on your trace map.

Complex keyword operators find segments where the specified key has been set, or not set.

Complex keyword operators

- none – The expression is true if the keyword is set. If the keyword is of boolean type, it will evaluate to the boolean value.
- `!` – The expression is true if the keyword is not set. If the keyword is of boolean type, it will evaluate to the boolean value.
- `=,!=` – Compare the value of the keyword.
- `edge(`*`source`*`, `*`destination`*`) {`*`filter`*`}` – Connection between services *`source`* and *`destination`*. Optional curly braces can contain a filter expression that applies to segments on this connection.
- `annotation[`*`key`*`]` – Value of an annotation with field *`key`*. The value of an annotation can be a Boolean, number, or string, so you can use any of the comparison operators of those types. You can use this keyword in combination with the `service` or `edge` keyword.
- `json` – JSON root cause object. See [Getting data from AWS X-Ray]() for steps to create JSON entities programmatically.

Use the service keyword to find traces for requests that hit a certain node on your trace map.

Example – Service filter

Requests that included a call to `api.example.com` with a fault (500 series error).

```
service("api.example.com") { fault }
```

You can exclude the service name to apply a filter expression to all nodes on your service map.

Example – service filter

Requests that caused a fault anywhere on your trace map.

```
service() { fault }
```

The edge keyword applies a filter expression to a connection between two nodes.

Example – edge filter

Request where the service `api.example.com` made a call to `backend.example.com` that failed with an error.

```
edge("api.example.com", "backend.example.com") { error }
```

You can also use the `!` operator with service and edge keywords to exclude a service or edge from the results of another filter expression.

Example – service and request filter

Request where the URL begins with `http://api.example.com/` and contains `/v2/` but does not reach a service named `api.example.com`.

```
http.url BEGINSWITH "http://api.example.com/" AND http.url CONTAINS "/v2/" AND !
service("api.example.com")
```

Example – service and response time filter

Find traces where `http url` is set and response time is greater than 2 seconds.

```
http.url AND responseTime > 2
```

For annotations, you can call all traces where annotation[*key*] is set, or use the comparison operators that correspond to the type of value.

Example – annotation with string value

Requests with an annotation named `gameid` with string value `"817DL6VO"`.

```
annotation[gameid] = "817DL6VO"
```

Example – annotation is set

Requests with an annotation named age set.

```
annotation[age]
```

Example – annotation is not set

Requests without an annotation named age set.

```
!annotation[age]
```

Example – annotation with number value

Requests with annotation age with numerical value greater than 29.

```
annotation[age] > 29
```

Example – annotation in combination with service or edge

```
service { annotation[request.id] = "917DL6VO" }
```

```
edge { source.annotation[request.id] = "916DL6VO" }
```

```
edge { destination.annotation[request.id] = "918DL6VO" }
```

Example – group with user

Requests where traces meet the `high_response_time` group filter (e.g. `responseTime > 3`), and the user is named Alice.

```
group.name = "high_response_time" AND user = "alice"
```

Example – JSON with root cause entity

Requests with matching root cause entities.

```
rootcause.json = #[{ "Services": [ { "Name": "GetWeatherData", "EntityPath": [{ "Name":
 "GetWeatherData" }, { "Name": "get_temperature" } ] }, { "Name": "GetTemperature",
 "EntityPath": [ { "Name": "GetTemperature" } ] } ] }]
```

id function

When you provide a service name to the `service` or `edge` keyword, you get results for all nodes that have that name. For more precise filtering, you can use the `id` function to specify a service type in addition to a name to distinguish between nodes with the same name.

Use the `account.id` function to specify a particular account for the service, when viewing traces from multiple accounts in a monitoring account.

```
id(name: "service-name", type:"service::type", account.id:"account-ID")
```

You can use the `id` function in place of a service name in service and edge filters.

```
service(id(name: "service-name", type:"service::type")) { filter }
```

```
edge(id(name: "service-one", type:"service::type"), id(name: "service-two",
 type:"service::type")) { filter }
```

For example, AWS Lambda functions result in two nodes in the trace map; one for the function invocation, and one for the Lambda service. The two nodes have the same name but different types. A standard service filter will find traces for both.

Example – service filter

Requests that include an error on any service named `random-name`.

```
service("random-name") { error }
```

Use the `id` function to narrow the search to errors on the function itself, excluding errors from the service.

Example – service filter with id function

Requests that include an error on a service named `random-name` with type `AWS::Lambda::Function`.

```
service(id(name: "random-name", type: "AWS::Lambda::Function")) { error }
```

To search for nodes by type, you can also exclude the name entirely.

Example – service filter with id function and service type

Requests that include an error on a service with type `AWS::Lambda::Function`.

```
service(id(type: "AWS::Lambda::Function")) { error }
```

To search for nodes for a particular AWS account, specify an account ID.

Example – service filter with id function and account ID

Requests that include a service within a specific account ID `AWS::Lambda::Function`.

```
service(id(account.id: "account-id"))
```

Cross-account tracing

AWS X-Ray supports *cross-account observability*, enabling you to monitor and troubleshoot applications that span multiple accounts within an AWS Region. You can seamlessly search, visualize, and analyze your metrics, logs, and traces in any of the linked accounts as if you were operating in a single account. This provides a complete view of requests that travel across multiple accounts. You can view cross-account traces in the X-Ray trace map and traces pages within the CloudWatch console.

The shared observability data can include any of the following types of telemetry:

- Metrics in Amazon CloudWatch
- Log groups in Amazon CloudWatch Logs
- Traces in AWS X-Ray
- Applications in Amazon CloudWatch Application Insights

Configure cross-account observability

To turn on cross-account observability, set up one or more AWS *monitoring* accounts and link them with multiple *source* accounts. A monitoring account is a central AWS account that can view and interact with observability data that's generated from source accounts. A source account is an individual AWS account that generates observability data for the resources that it contains.

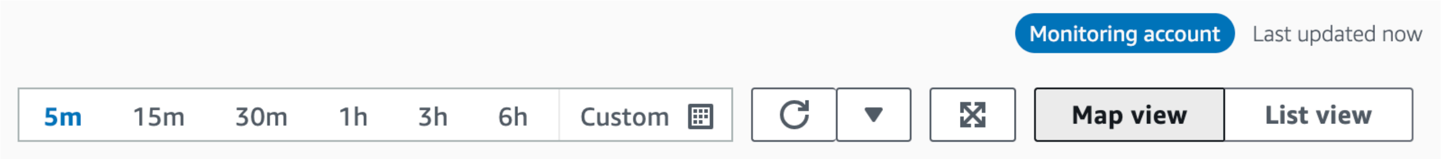
Source accounts share their observability data with monitoring accounts. Traces are copied from each source account to up to five monitoring accounts. Copies of traces from source accounts to the first monitoring account are free. Copies of traces sent to additional monitoring accounts are charged to each source account, based on standard pricing. For more information, see AWS X-Ray pricing and Amazon CloudWatch pricing.

To create links between monitoring accounts and source accounts, use the CloudWatch console or the new Observability Access Manager commands in the AWS CLI and API. For more information, see CloudWatch cross-account observability.

> **Note**
>
> X-Ray traces are billed to the AWS account where they're received. If a sampled request spans services across more than one AWS account, each account records a separate trace, and all traces share the same trace ID. To learn more about cross-account observability pricing, see AWS X-Ray pricing and Amazon CloudWatch pricing.

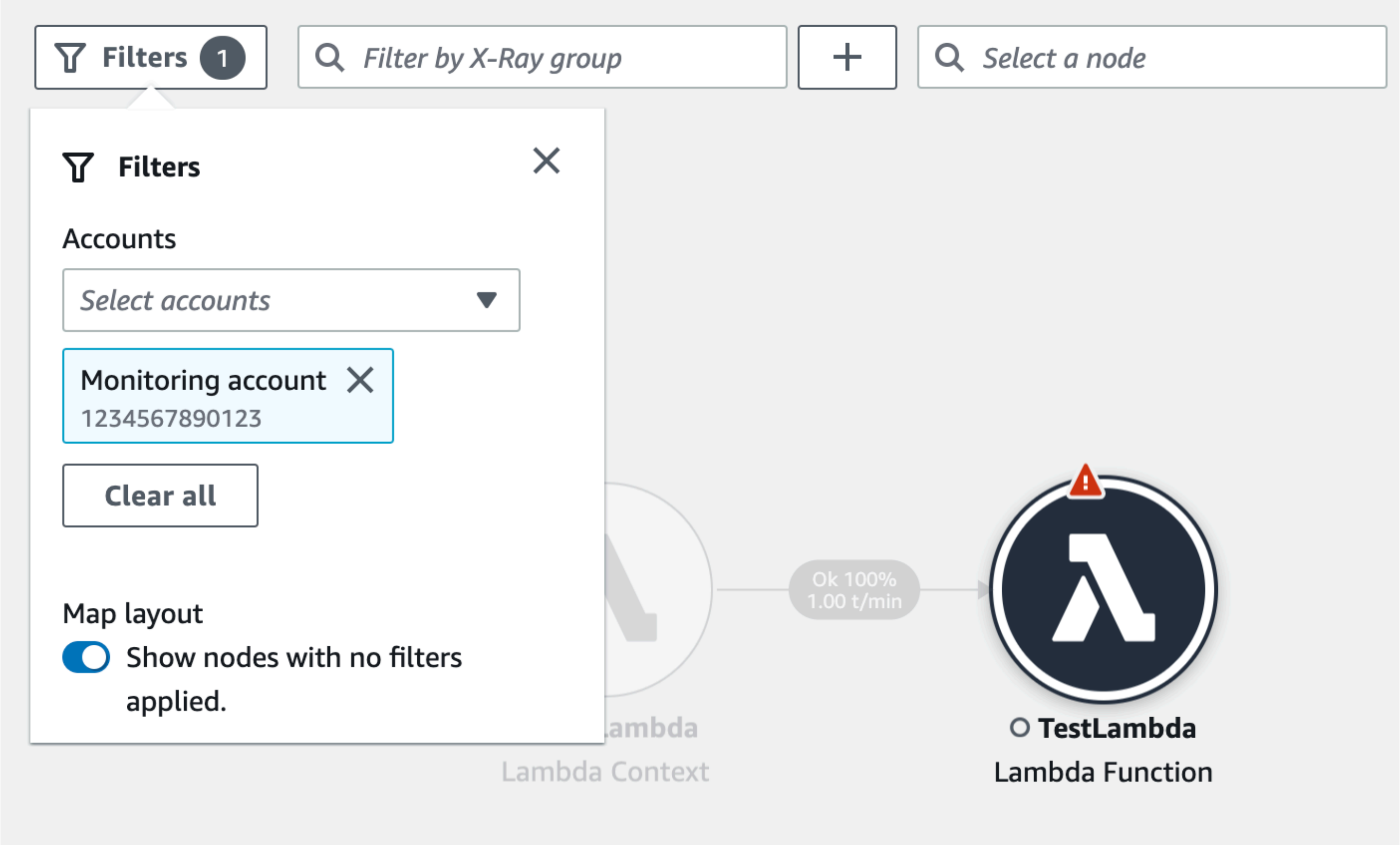
Viewing cross-account traces

Cross-account traces are displayed in the monitoring account. Each source account displays only local traces for that specific account. The following sections assume that you're signed in to the monitoring account and have opened the Amazon CloudWatch console. On both the trace map and traces pages, a monitoring account badge is displayed in the upper-right corner.

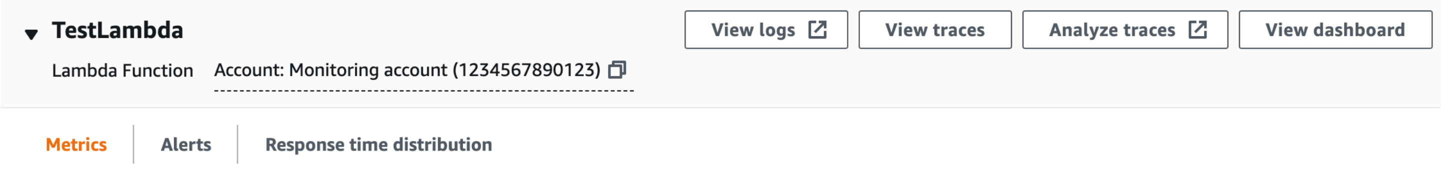


Trace map

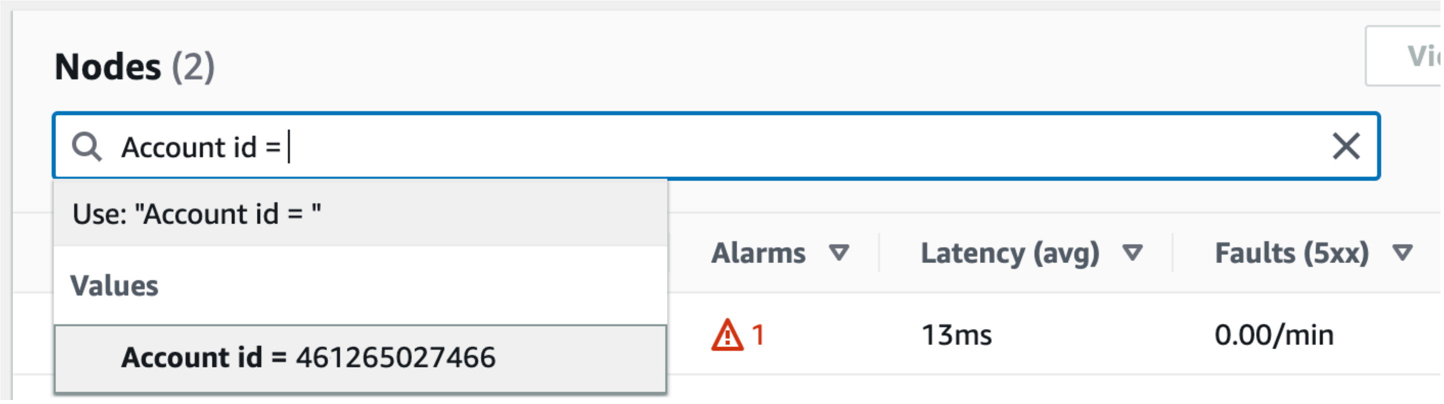
In the CloudWatch console, choose **Trace Map** under **X-Ray traces** from the left navigation pane. By default, the trace map displays nodes for all source accounts that send traces to the monitoring account, and nodes for the monitoring account itself. On the trace map, choose **Filters** from the upper left to filter the trace map using the **Accounts** drop-down. After an account filter is applied, service nodes from accounts that don't match the current filter are grayed out.



When you choose a service node, the node details pane includes the service's account ID and label.



In the upper-right corner of the trace map, choose **List view** to see a list of service nodes. The list of service nodes includes services from the monitoring account and all configured source accounts. Filter the list of nodes by **Account label** or **Account id** by choosing them from the **Nodes** filter.

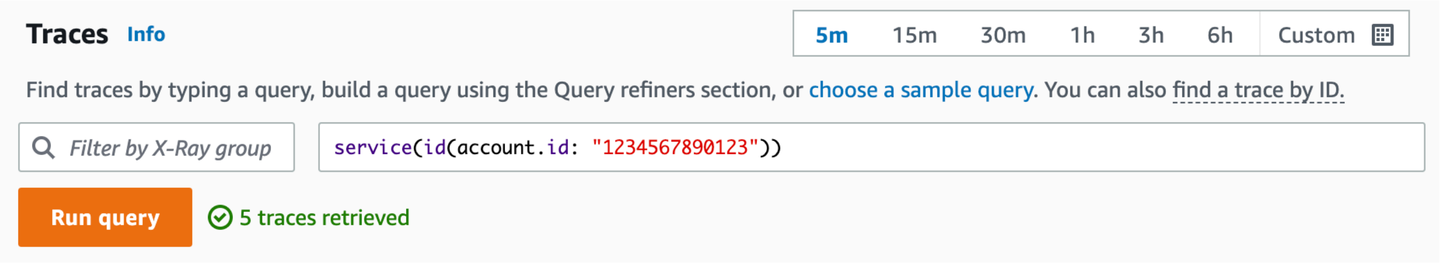


Traces

View trace details for traces that span multiple accounts by opening the CloudWatch console from the monitoring account, and choosing **Traces** under **X-Ray traces** in the left navigation pane. You can also open this page by choosing a node in the X-Ray **Trace Map**, and then choosing **View traces** from the node details pane.

The **Traces** page supports querying by account ID. To get started, enter a query that includes one or more account IDs. The following example queries for traces that have passed through account ID X or *Y*:

```
service(id(account.id:"X")) OR service(id(account.id:"Y"))
```

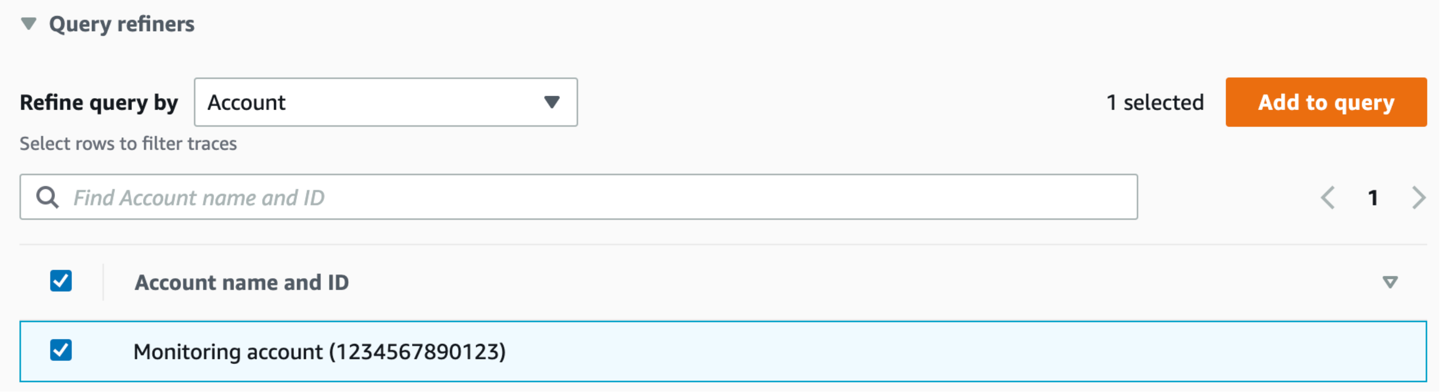


Find traces by typing a query, build a query using the Query refiners section, or choose a sample query. You can also find a trace by ID.

```
service(id(account.id: "1234567890123"))
```

5 traces retrieved

Refine your query by **Account**. Select one or more accounts from the list, and choose **Add to query**.



Refine query by: Account

Select rows to filter traces

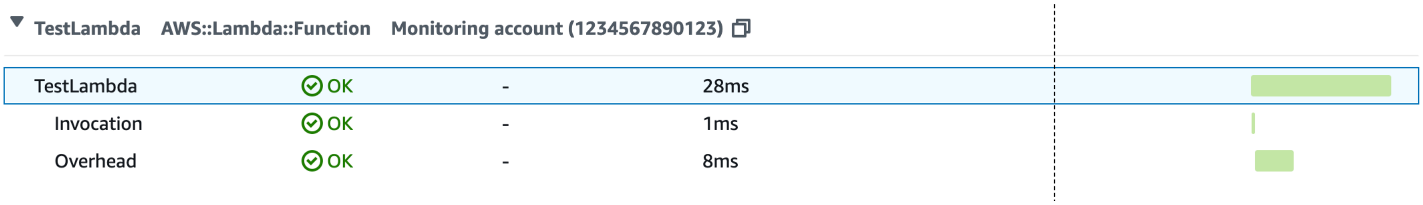
1 selected

☑	Account name and ID
☑	Monitoring account (1234567890123)

Trace details

View details for a trace by choosing it from the **Traces** list at the bottom of the **Traces** page. The **Trace details** displays, including a trace details map with service nodes from across all accounts that the trace passed through. Choose a specific service node to see its corresponding account.

The **Segments timeline** section displays the account details for each segment in the timeline.



▼ TestLambda AWS::Lambda::Function Monitoring account (1234567890123)

TestLambda	✓ OK	-	28ms
Invocation	✓ OK	-	1ms
Overhead	✓ OK	-	8ms

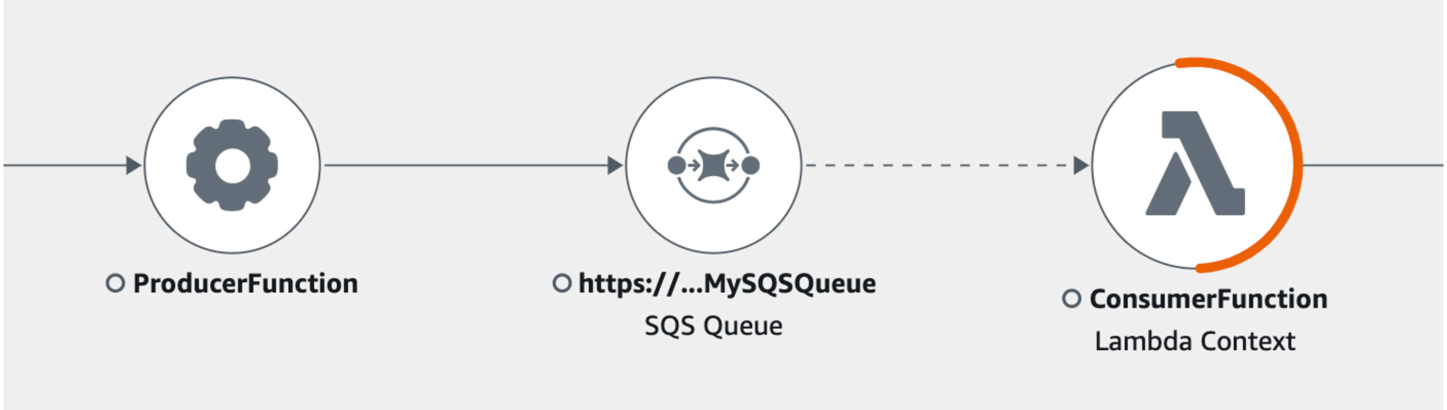
Tracing event-driven applications

AWS X-Ray supports tracing event-driven applications using Amazon SQS and AWS Lambda. Use the CloudWatch console to see a connected view of each request as it's queued with Amazon SQS and processed by one or more Lambda functions. Traces from upstream message producers are automatically linked to traces from downstream Lambda consumer nodes, creating an end-to-end view of the application.

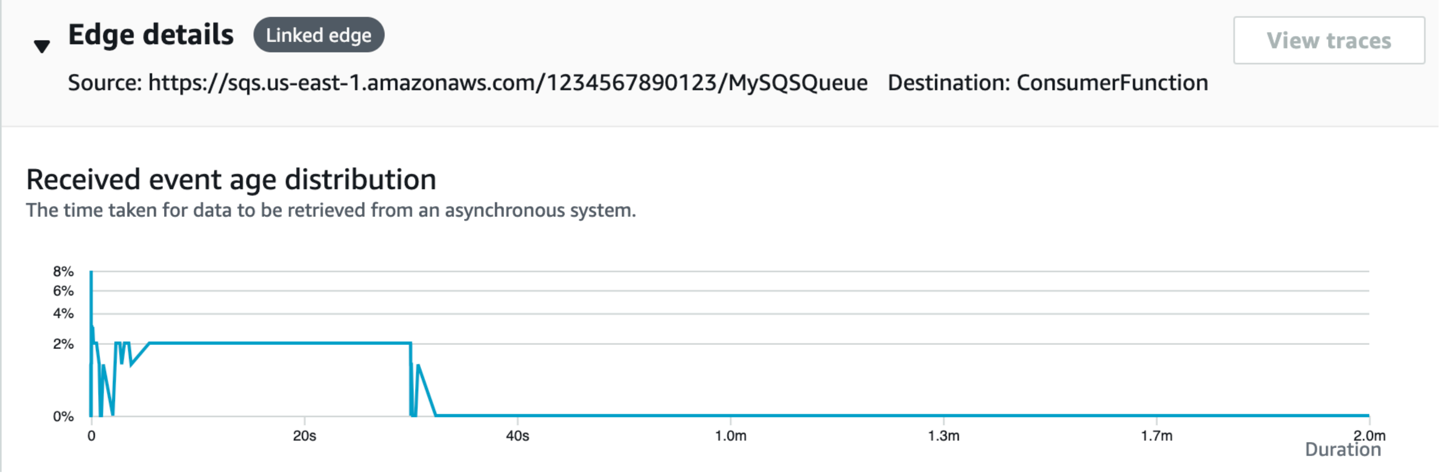
> **Note**
>
> Each trace segment can be linked to up to 20 traces, while a trace can include a maximum of 100 links. In certain scenarios, linking additional traces may result in exceeding the maximum trace document size, causing a potentially incomplete trace. This can happen, for example, when a Lambda function with tracing enabled sends many SQS messages to a queue in a single invocation. If you encounter this issue, a mitigation is available which uses the X-Ray SDKs. See the X-Ray SDK for Java, Node.js, Python, Go, or .NET for more information.

View linked traces in the trace map

Use the **Trace Map** page within the CloudWatch console to view a trace map with traces from message producers that are linked to traces from Lambda consumers. These links are displayed with a dashed-line edge that connects the Amazon SQS node and downstream Lambda consumer nodes.



Select a dashed-line edge to display a *received event age* histogram, which maps the spread of event age when it's received by consumers. The age is calculated each time an event is received.



Edge details Linked edge

View traces

Source: https://sqs.us-east-1.amazonaws.com/1234567890123/MySQSQueue Destination: ConsumerFunction

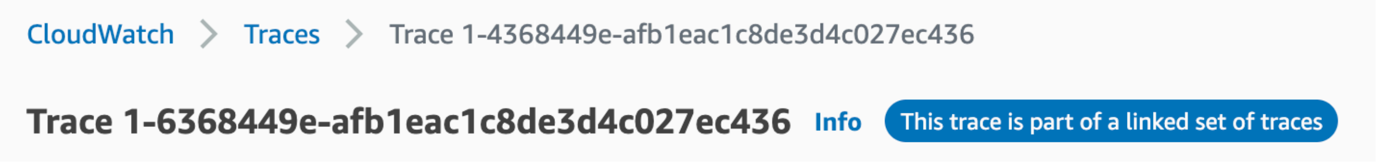
Received event age distribution

The time taken for data to be retrieved from an asynchronous system.

View linked trace details

View trace details sent from a message producer, Amazon SQS queue, or Lambda consumer:

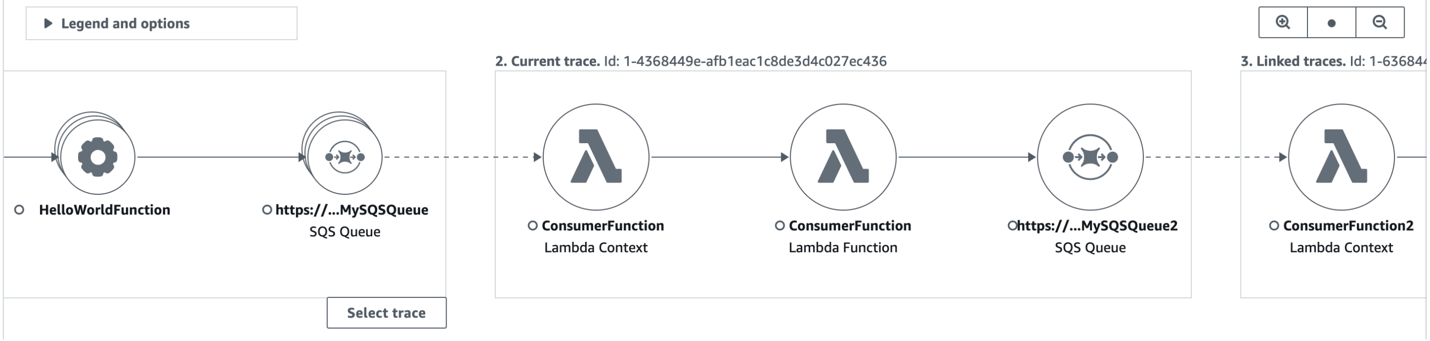
1. Use the **Trace Map** to select a message producer, Amazon SQS, or Lambda consumer node.
2. Choose **View traces** from the node details pane to display a list of traces. You can also navigate directly to the **Traces** page within the CloudWatch console.
3. Choose a specific trace from the list to open the trace details page. The trace details page displays a message when the selected trace is part of a linked set of traces.



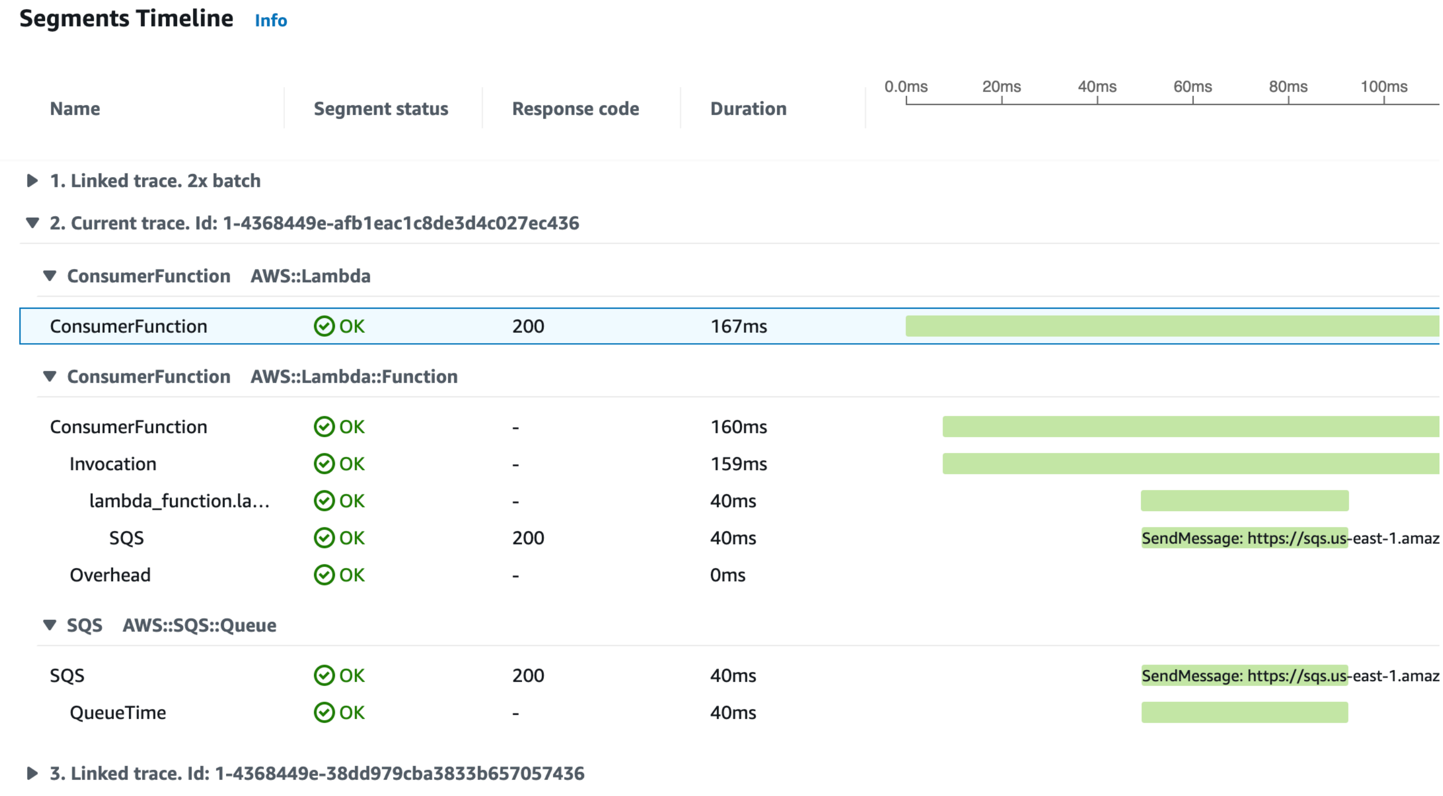
CloudWatch > Traces > Trace 1-4368449e-afb1eac1c8de3d4c027ec436

Trace 1-6368449e-afb1eac1c8de3d4c027ec436 Info This trace is part of a linked set of traces

The trace details map displays the current trace, along with upstream and downstream linked traces, each of which are contained within a box that indicates the bounds of each trace. If the currently selected trace is linked to multiple upstream or downstream traces, the nodes within the upstream or downstream linked traces are stacked, and a **Select trace** button is displayed.



Beneath the trace details map, a timeline of trace segments displays, including upstream and downstream linked traces. If there are multiple upstream or downstream linked traces, their segment details can't be displayed. To view segment details for a single trace within a set of linked traces, select a single trace as described below.



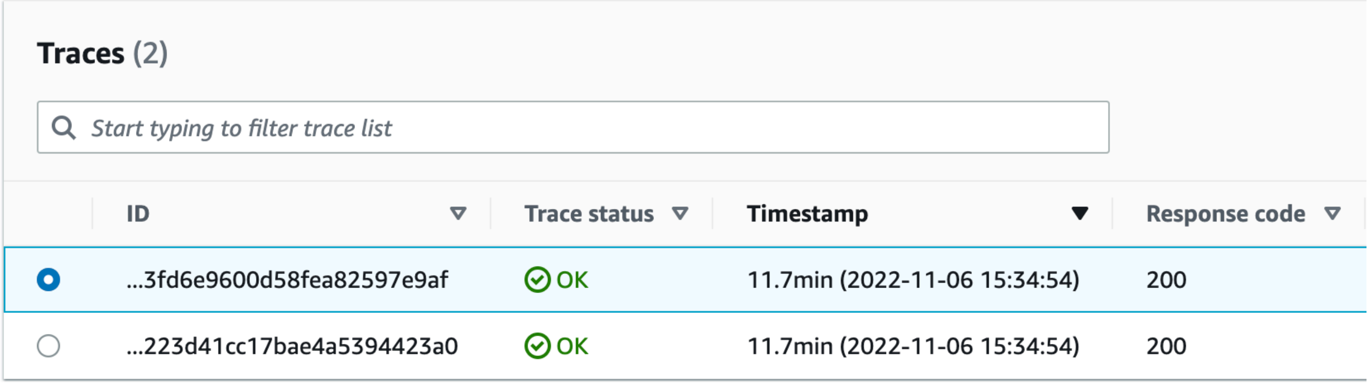
Segments Timeline Info

Name	Segment status	Response code	Duration	0.0ms 20ms 40ms 60ms 80ms 100ms
▶ 1. Linked trace. 2x batch				
▼ 2. Current trace. Id: 1-4368449e-afb1eac1c8de3d4c027ec436				
▼ ConsumerFunction AWS::Lambda				
ConsumerFunction	✓ OK	200	167ms	
▼ ConsumerFunction AWS::Lambda::Function				
ConsumerFunction	✓ OK	-	160ms	
Invocation	✓ OK	-	159ms	
lambda_function.la...	✓ OK	-	40ms	
SQS	✓ OK	200	40ms	SendMessage: https://sqs.us-east-1.amaz
Overhead	✓ OK	-	0ms	
▼ SQS AWS::SQS::Queue				
SQS	✓ OK	200	40ms	SendMessage: https://sqs.us-east-1.amaz
QueueTime	✓ OK	-	40ms	
▶ 3. Linked trace. Id: 1-4368449e-38dd979cba3833b657057436				

Select a single trace within a set of linked traces

Filter a linked set of traces to a single trace, to see segment details in the timeline.

1. Choose **Select trace** underneath the linked traces on the trace details map. A list of traces displays.

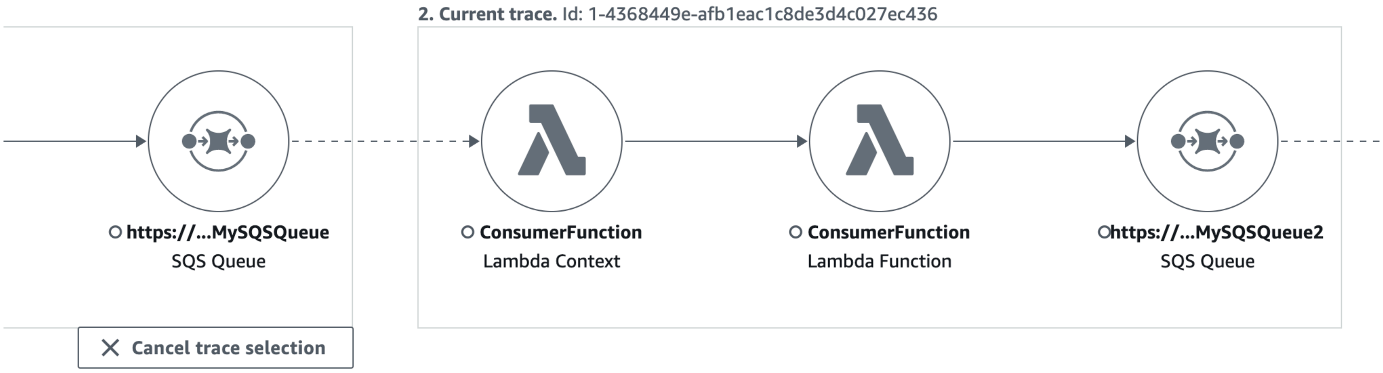


Traces (2)

Start typing to filter trace list

	ID	Trace status	Timestamp	Response code
●	...3fd6e9600d58fea82597e9af	✓ OK	11.7min (2022-11-06 15:34:54)	200
○	...223d41cc17bae4a5394423a0	✓ OK	11.7min (2022-11-06 15:34:54)	200

2. Select the radio button next to a trace to view it within the trace details map.
3. Choose **Cancel trace selection** to view the entire set of linked traces.



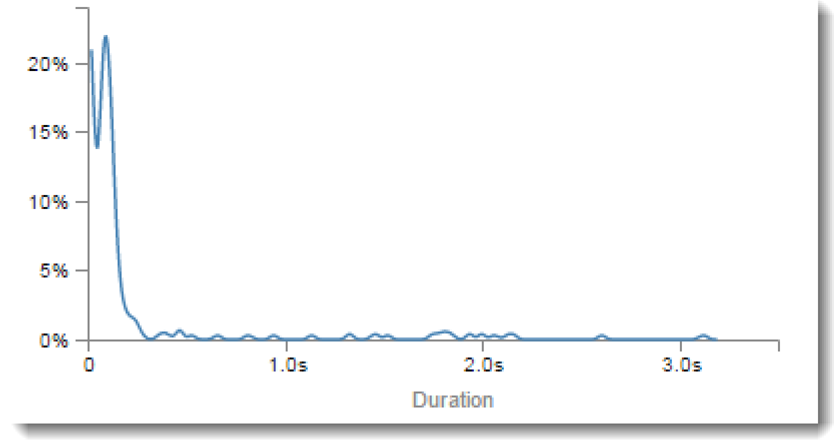
Using latency histograms

When you select a node or edge on an AWS X-Ray trace map, the X-Ray console shows a latency distribution histogram.

Latency

Latency is the amount of time between when a request starts and when it completes. A histogram shows a distribution of latencies. It shows duration on the x-axis, and the percentage of requests that match each duration on the y-axis.

This histogram shows a service that completes most requests in less than 300 ms. A small percentage of requests take up to 2 seconds, and a few outliers take more time.



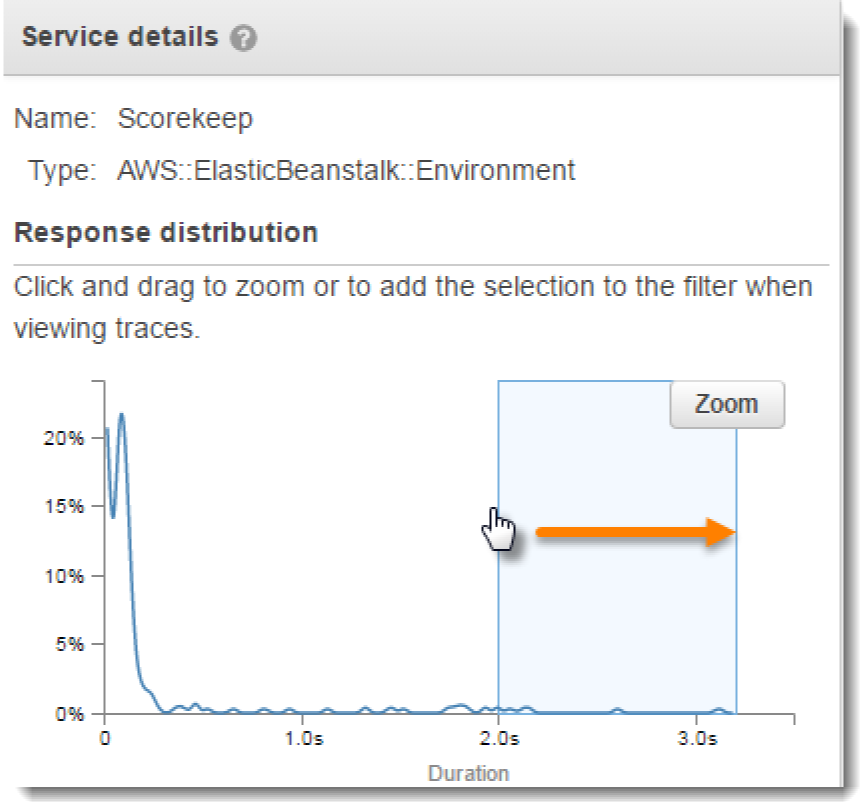
Interpreting service details

Service histograms and edge histograms provide a visual representation of latency from the viewpoint of a service or requester.

- Choose a *service node* by clicking the circle. X-Ray shows a histogram for requests served by the service. The latencies are those recorded by the service, and don't include any network latency between the service and the requester.
- Choose an *edge* by clicking the line or arrow tip of the edge between two services. X-Ray shows a histogram for requests from the requester that were served by the downstream service. The latencies are those recorded by the requester, and include latency in the network connection between the two services.

To interpret the **Service details** panel histogram, you can look for values that differ the most from the majority of values in the histogram. These *outliers* can be seen as peaks or spikes in the histogram, and you can view the traces for a specific area to investigate what's going on.

To view traces filtered by latency, select a range on the histogram. Click where you want to start the selection and drag from left to right to highlight a range of latencies to include in the trace filter.



Service details

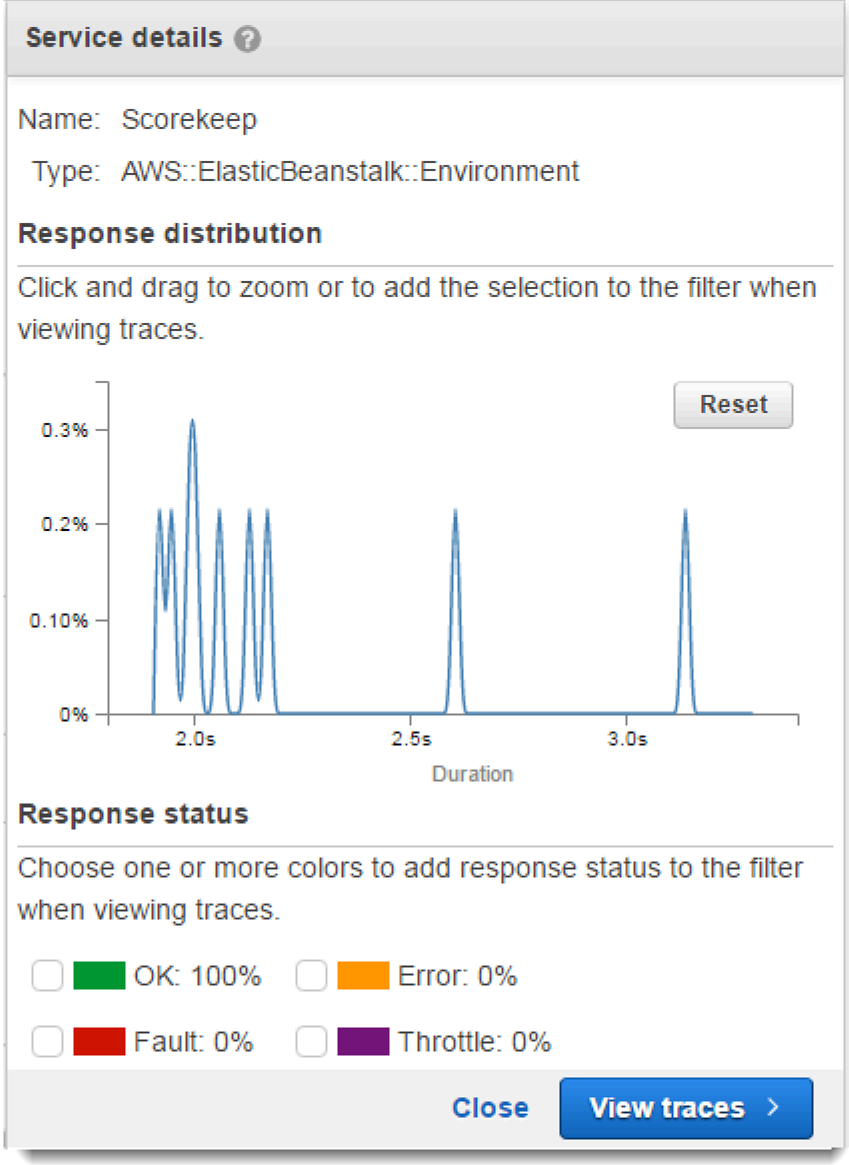
Name: Scorekeep

Type: AWS::ElasticBeanstalk::Environment

Response distribution

Click and drag to zoom or to add the selection to the filter when viewing traces.

After selecting a range, you can choose **Zoom** to view just that portion of the histogram and refine your selection.



Service details

Name: Scorekeep

Type: AWS::ElasticBeanstalk::Environment

Response distribution

Click and drag to zoom or to add the selection to the filter when viewing traces.

Response status

Choose one or more colors to add response status to the filter when viewing traces.

□ 🟩 OK: 100% □ 🟧 Error: 0%

□ 🟥 Fault: 0% □ 🟪 Throttle: 0%

Once you have the focus set to the area you're interested in, choose **View traces**.

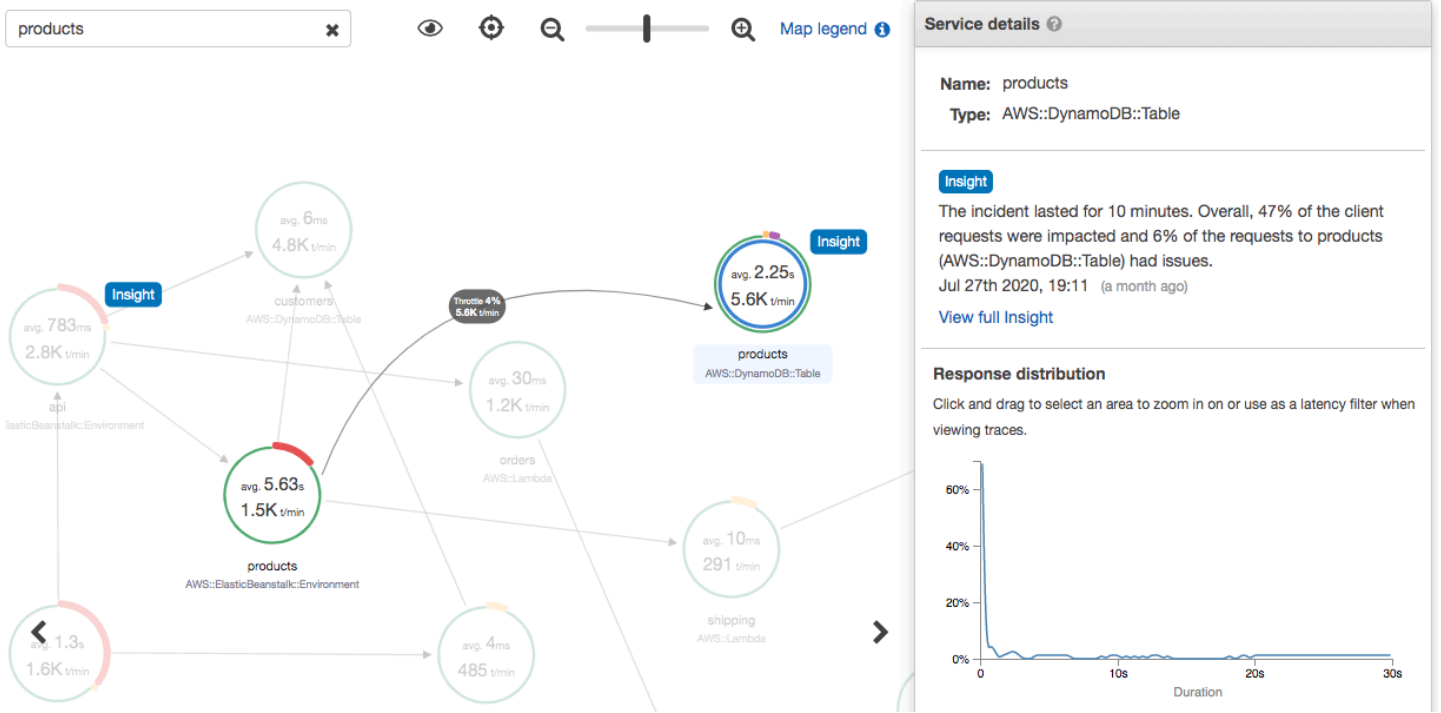
Using X-Ray insights

AWS X-Ray continuously analyzes trace data in your account to identify emergent issues in your applications. When fault rates exceed the expected range, it creates an *insight* that records the issue and tracks its impact until it's resolved. With insights, you can:

- Identify where in your application issues are occurring, the root cause of the issue, and associated impact. The impact analysis provided by insights enables you to derive the severity and priority of an issue.

- Receive notifications as the issue changes over time. Insights notifications can be integrated with your monitoring and alerting solution using Amazon EventBridge. This integration enables you to send automated emails or alerts based on the severity of the issue.

The X-Ray console identifies nodes with ongoing incidents in the trace map. To see a summary of the insight, choose the affected node. You can also view and filter insights by choosing **Insights** from the navigation pane on the left.



Service details

Name: products
Type: AWS::DynamoDB::Table

Insight

The incident lasted for 10 minutes. Overall, 47% of the client requests were impacted and 6% of the requests to products (AWS::DynamoDB::Table) had issues.

Jul 27th 2020, 19:11 (a month ago)

View full Insight

Response distribution

Click and drag to select an area to zoom in on or use as a latency filter when viewing traces.

X-Ray creates an insight when it detects an *anomaly* in one or more nodes of the service map. The service uses statistical modeling to predict the expected fault rates of services in your application. In the preceding example, the anomaly is an increase in faults from AWS Elastic Beanstalk. The Elastic Beanstalk server experienced multiple API call timeouts, causing an anomaly in the downstream nodes.

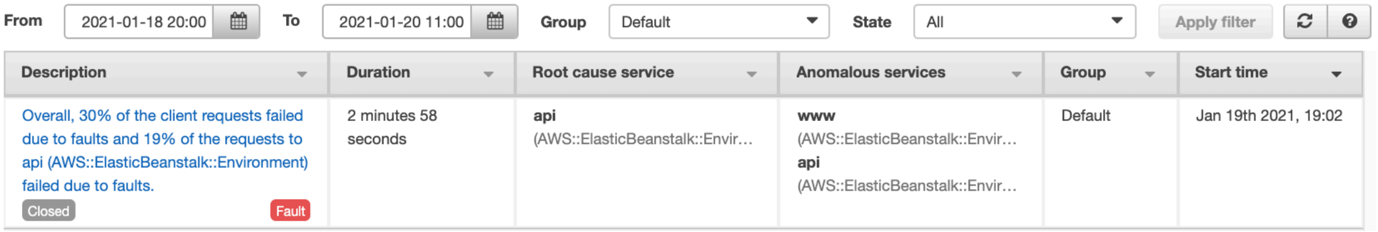
Enable insights in the X-Ray console

Insights must be enabled for each group you want to use insights features with. You can enable insights from the **Groups** page.

1. Open the X-Ray console.

2. Select an existing group or create a new one by choosing **Create group**, and then select **Enable Insights**. For more information about configuring groups in the X-Ray console, see Configuring groups.

3. In the navigation pane on the left, choose **Insights**, and then choose an insight to view.



From 2021-01-18 20:00 To 2021-01-20 11:00 Group Default State All

Description	Duration	Root cause service	Anomalous services	Group	Start time
Overall, 30% of the client requests failed due to faults and 19% of the requests to api (AWS::ElasticBeanstalk::Environment) failed due to faults. Closed Fault	2 minutes 58 seconds	api (AWS::ElasticBeanstalk::Envir...	www (AWS::ElasticBeanstalk::Envir... api (AWS::ElasticBeanstalk::Envir...	Default	Jan 19th 2021, 19:02

> **Note**
>
> X-Ray uses GetInsightSummaries, GetInsight, GetInsightEvents, and GetInsightImpactGraph API operations to retrieve data from insights.
> For more information, see How AWS X-Ray works with IAM.

Enable insights notifications

With insights notifications, a notification is created for each insight event, such as when an insight is created, changes significantly, or is closed. Customers can receive these notifications through Amazon EventBridge events, and use conditional rules to take actions such as SNS notification, Lambda invocation, posting messages to an SQS queue, or any of the targets EventBridge supports. Insights notifications are emitted on a best-effort basis but are not guaranteed. For more information about targets, see Amazon EventBridge Targets.

You can enable insights notifications for any insights enabled group from the **Groups** page.

To enable notifications for an X-Ray group

1. Open the X-Ray console.

2. Select an existing group or create a new one by choosing **Create group**, ensure that **Enable Insights** is selected, and then select **Enable Notifications**. For more information about configuring groups in the X-Ray console, see Configuring groups.

To configure Amazon EventBridge conditional rules

1. Open the Amazon EventBridge console.
2. Navigate to **Rules** in the left navigation bar, and choose **Create rule**.
3. Provide a name and description for the rule.
4. Choose **Event pattern**, and then choose **Custom pattern**. Provide a pattern containing `"source": [ "aws.xray" ]` and `"detail-type": [ "AWS X-Ray Insight Update" ]`. The following are some examples of possible patterns.
   - Event pattern to match all incoming events from X-Ray insights:

     ```
     {
     "source": [ "aws.xray" ],
     "detail-type": [ "AWS X-Ray Insight Update" ]
     }
     ```

   - Event pattern to match a specified **`state`** and **`category`**:

     ```

     {
     "source": [ "aws.xray" ],
     "detail-type": [ "AWS X-Ray Insight Update" ],
     "detail": {
             "State": [ "ACTIVE" ],
             "Category": [ "FAULT" ]
       }
     }

     ```

5. Select and configure the targets that you would like to invoke when an event matches this rule.
6. (Optional) Provide tags to more easily identify and select this rule.
7. Choose **Create**.

> **Note**
>
> X-Ray insights notifications sends events to Amazon EventBridge, which does not currently support customer managed keys. For more information, see Data protection in AWS X-Ray.

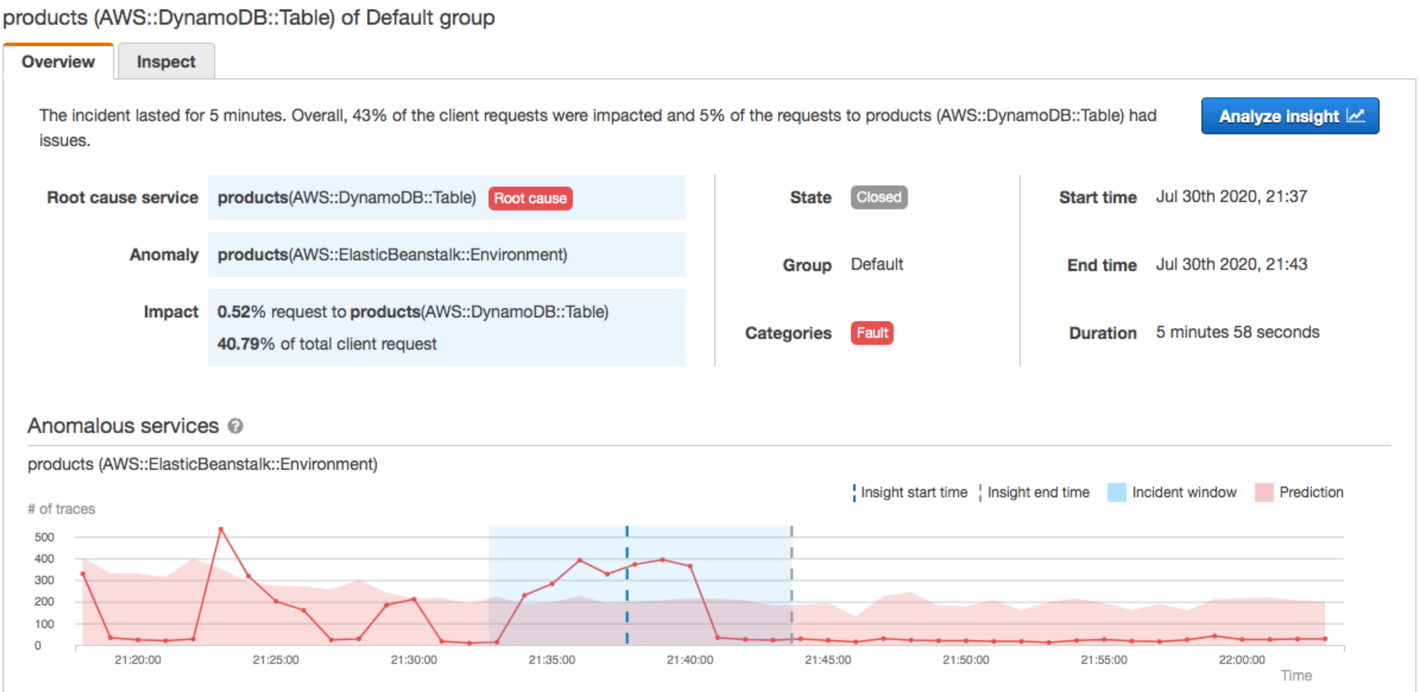
Insight overview

The overview page for an insight attempts to answer three key questions:

- What is the underlying issue?
- What is the root cause?
- What is the impact?

The **Anomalous services** section shows a timeline for each service that illustrates the change in fault rates during the incident. The timeline shows the number of traces with faults overlaid on a solid band that indicates the expected number of faults based on the amount of traffic recorded. The duration of the insight is visualized by the *Incident window*. The incident window begins when X-Ray observes the metric becoming anomalous and persists while the insight is active.

The following example shows an increase in faults that caused an incident:



products (AWS::DynamoDB::Table) of Default group

The incident lasted for 5 minutes. Overall, 43% of the client requests were impacted and 5% of the requests to products (AWS::DynamoDB::Table) had issues.

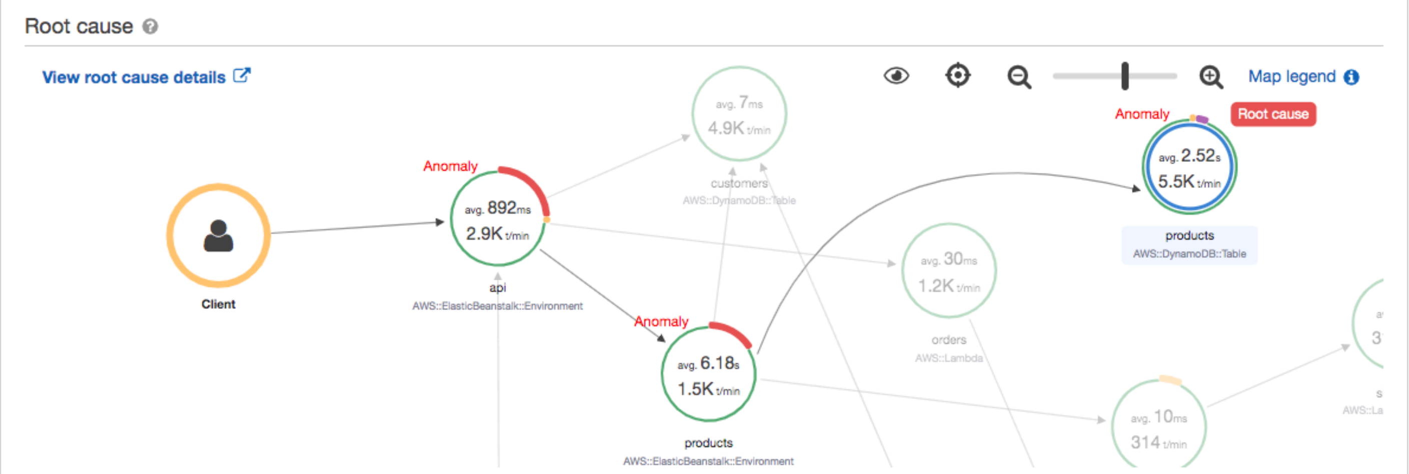
Root cause service	**products**(AWS::DynamoDB::Table) Root cause
Anomaly	**products**(AWS::ElasticBeanstalk::Environment)
Impact	**0.52%** request to **products**(AWS::DynamoDB::Table) **40.79%** of total client request
State	Closed
Group	Default
Categories	Fault
Start time	Jul 30th 2020, 21:37
End time	Jul 30th 2020, 21:43
Duration	5 minutes 58 seconds

Anomalous services

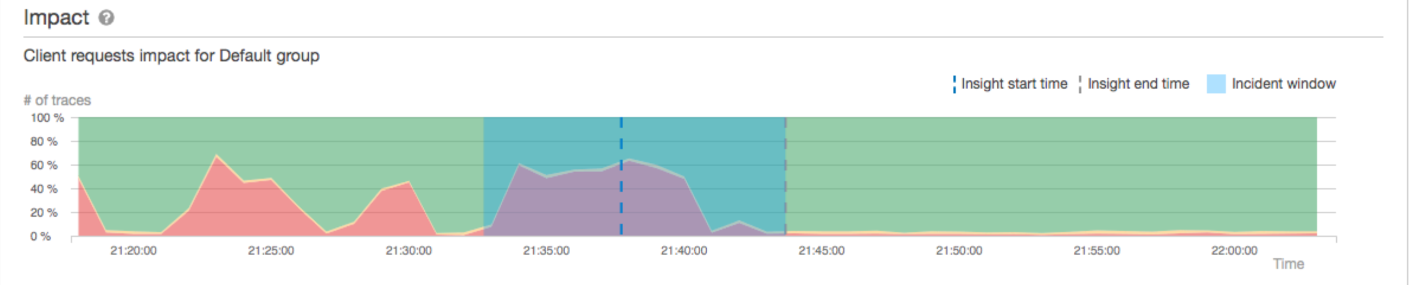
products (AWS::ElasticBeanstalk::Environment)

The **Root cause** section shows a trace map focused on the root cause service and the impacted path. You may hide the unaffected nodes by selecting the eye icon in the top right of the Root cause map. The root cause service is the farthest downstream node where X-Ray identified an anomaly. It can represent a service that you instrumented or an external service that your service called with an instrumented client. For example, if you call Amazon DynamoDB with an instrumented AWS SDK client, an increase in faults from DynamoDB results in an insight with DynamoDB as the root cause.

To further investigate the root cause, select **View root cause details** on the root cause graph. You can use the **Analytics** page to investigate the root cause and related messages. For more information, see [Interacting with the Analytics console](#).



Faults that continue upstream in the map can impact multiple nodes and cause multiple anomalies. If a fault is passed all the way back to the user that made the request, the result is a *client fault*. This is a fault in the root node of the trace map. The **Impact** graph provides a timeline of the client experience for the entire group. This experience is calculated based on percentages of the following states: **Fault**, **Error**, **Throttle**, and **Okay**.



This example shows an increase in traces with a fault at the root node during the time of an incident. Incidents in downstream services don't always correspond to an increase in client errors.

Choosing **Analyze insight** opens the X-Ray Analytics console in a window where you can dive deep into the set of traces causing the insight. For more information, see [Interacting with the Analytics console](#).

Understanding impact

AWS X-Ray measures the impact caused by an ongoing issue as part of generating insights and notifications. The impact is measured in two ways:

- Impact to the X-Ray group
- Impact on the root cause service

This impact is determined by the percentage of request that are failing or causing an error within a given time period. This impact analysis allows you to derive the severity and priority of the issue based on your particular scenario. This impact is available as part of the console experience in addition to insights notifications.

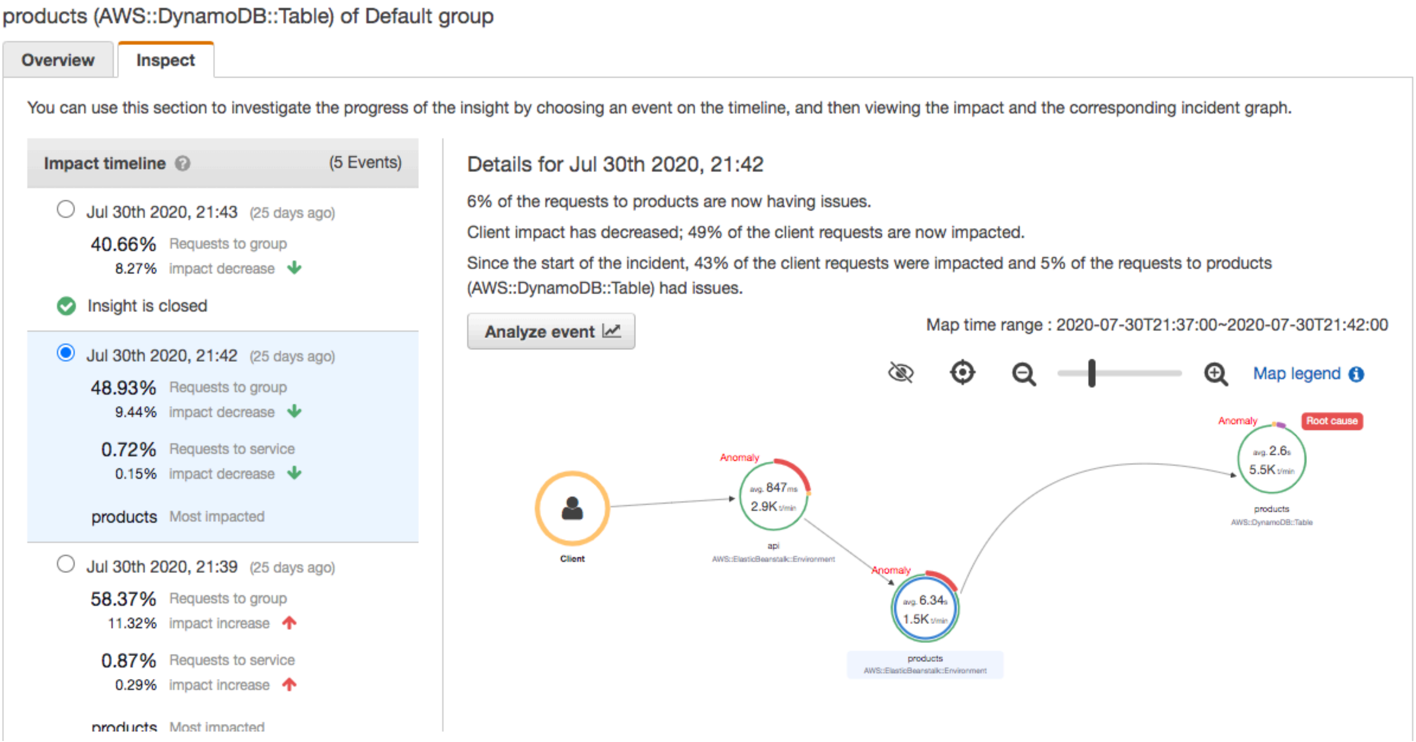
Deduplication

AWS X-Ray insights de-duplicates issues across multiple microservices. It uses anomaly detection to determine the service that is the root cause of an issue, determines if other related services are exhibiting anomalous behavior due to the same root cause, and records the result as a single insight.

Review an insight's progress

X-Ray reevaluates insights periodically until they are resolved, and records each notable intermediate change as a notification, which can be sent as an Amazon EventBridge event. This enables you to build processes and workflows to determine how the issue has changed over time, and take appropriate actions such as sending an email or integrating with an alerting system using EventBridge.

You can review incident events in the **Impact Timeline** on the **Inspect** page. By default the timeline displays the most impacted service until you choose a different service.



products (AWS::DynamoDB::Table) of Default group

Overview | Inspect

You can use this section to investigate the progress of the insight by choosing an event on the timeline, and then viewing the impact and the corresponding incident graph.

Impact timeline (5 Events)

○ Jul 30th 2020, 21:43 (25 days ago)

40.66% Requests to group
8.27% impact decrease ↓

✓ Insight is closed

● Jul 30th 2020, 21:42 (25 days ago)

48.93% Requests to group
9.44% impact decrease ↓

0.72% Requests to service
0.15% impact decrease ↓

products Most impacted

○ Jul 30th 2020, 21:39 (25 days ago)

58.37% Requests to group
11.32% impact increase ↑

0.87% Requests to service
0.29% impact increase ↑

products Most impacted

Details for Jul 30th 2020, 21:42

6% of the requests to products are now having issues.

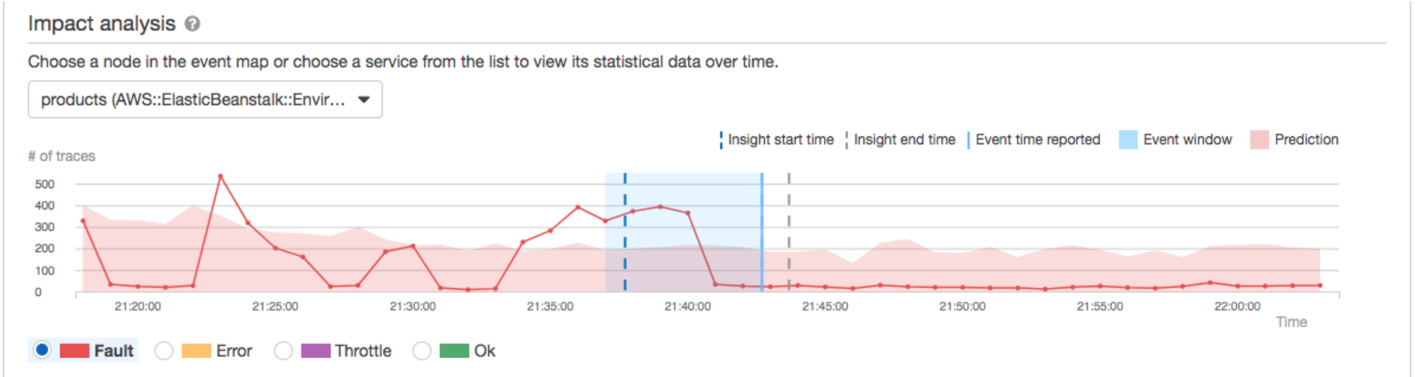
Client impact has decreased; 49% of the client requests are now impacted.

Since the start of the incident, 43% of the client requests were impacted and 5% of the requests to products (AWS::DynamoDB::Table) had issues.

Analyze event

Map time range : 2020-07-30T21:37:00~2020-07-30T21:42:00

To see a trace map and graphs for an event, choose it from the impact timeline. The trace map shows services in your application that are affected by the incident. Under **Impact analysis**, graphs show fault timelines for the selected node and for clients in the group.



Impact analysis

Choose a node in the event map or choose a service from the list to view its statistical data over time.

products (AWS::ElasticBeanstalk::Envir...

● Fault ○ Error ○ Throttle ○ Ok

To take a deeper look at the traces involved in an incident, choose **Analyze event** on the **Inspect** page. You can use the **Analytics** page to refine the list of traces and identify affected users. For more information, see Interacting with the Analytics console.

Interacting with the Analytics console

The AWS X-Ray Analytics console is an interactive tool for interpreting trace data to quickly understand how your application and its underlying services are performing. The console enables you to explore, analyze, and visualize traces through interactive response time and time-series graphs.

When making selections in the Analytics console, the console constructs filters to reflect the selected subset of all traces. You can refine the active dataset with increasingly granular filters by clicking the graphs and the panels of metrics and fields that are associated with the current trace set.

Topics

- Console features
- Response time distribution
- Time series activity
- Workflow examples
- Observe faults on the service graph
- Identify response time peaks
- View all traces marked with a status code
- View all items in a subgroup and associated to a user
- Compare two sets of traces with different criteria
- Identify a trace of interest and view its details

Console features

The X-Ray Analytics console uses the following key features for grouping, filtering, comparing, and quantifying trace data.

Features

Feature	Description
Groups	The initial selected group is `Default`. To change the retrieved group, select a different group from the menu to the right of the main filter expression search bar. To learn more about groups see, Using filter expressions with groups.

Feature	Description
Retrieved traces	By default, the Analytics console generates graphs based on all traces in the selected group. Retrieved traces represent all traces in your working set. You can find the trace count in this tile. Filter expressions you apply to the main search bar refine and update the retrieved traces.
Show in charts/Hide from charts	A toggle to compare the active group against the retrieved traces. To compare the data related to the group against any active filters, choose **Show in charts**. To remove this view from the charts, choose **Hide from charts**.
Filtered trace set A	Through interactions with the graphs and tables, apply filters to create the criteria for **Filtered trace set A**. As the filters are applied, the number of applicable traces and the percentage of traces from the total that are retrieved are calculated within this tile. Filters populate as tags within the **Filtered trace set A** tile and can also be removed from the tile.
Refine	This function updates the set of retrieved traces based on the filters applied to trace set A. Refining the retrieved trace set refreshes the working set of all traces retrieved based on the filters for trace set A. The working set of retrieved traces is a sampled subset of all traces in the group.

Feature	Description
Filtered trace set B	When created, **Filtered trace set B** is a copy of **Filtered trace set A**. To compare the two trace sets, make new filter selections that will apply to trace set B, while trace set A remains fixed. As the filters are applied, the number of applicable traces and the percentage of traces from the total retrieved are calculated within this tile. Filters populate as tags within the **Filtered trace set B** tile and can also be removed from the tile.
Response time root cause entity paths	A table of recorded entity paths. X-Ray determines which path in your trace is the most likely cause for the response time. The format indicates a hierarchy of entities that are encountered, ending in a response time root cause. Use these rows to filter for recurring response time faults. For more information about customizing a root cause filter and getting data through the API see, Retrieving and refining root cause analytics.
Delta (�)	A column that is added to the metrics tables when both trace set A and trace set B are active. The Delta column calculates the difference in percentage of traces between trace set A and trace set B.

Response time distribution

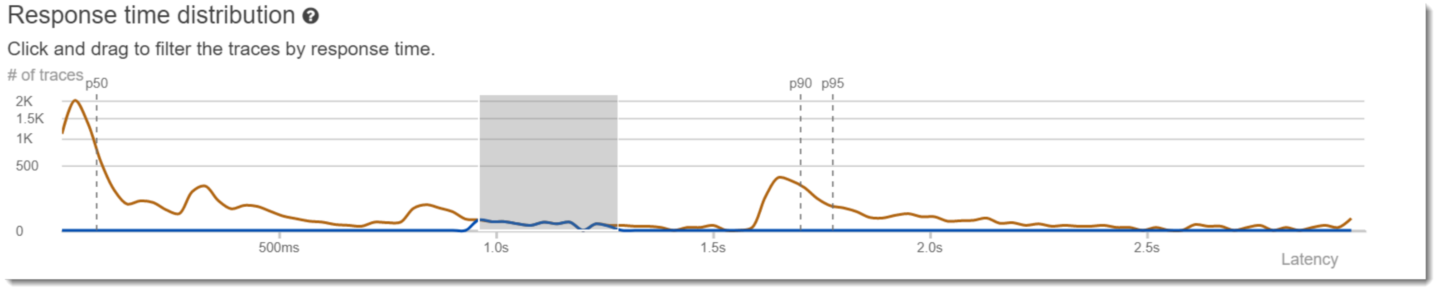
The X-Ray Analytics console generates two primary graphs to help you visualize traces: **Response Time Distribution** and **Time Series Activity**. This section and the following provide examples of each, and explain the basics of how to read the graphs.

The following are the colors associated with the response time line graph (the time series graph uses the same color scheme):

- **All traces in the group** – gray
- **Retrieved traces** – orange
- **Filtered trace set A** – green
- **Filtered trace set B** – blue

Example – Response time distribution

The response time distribution is a chart that shows the number of traces with a given response time. Click and drag to make selections within the response time distribution. This selects and creates a filter on the working trace set named `responseTime` for all traces within a specific response time.



Response time distribution

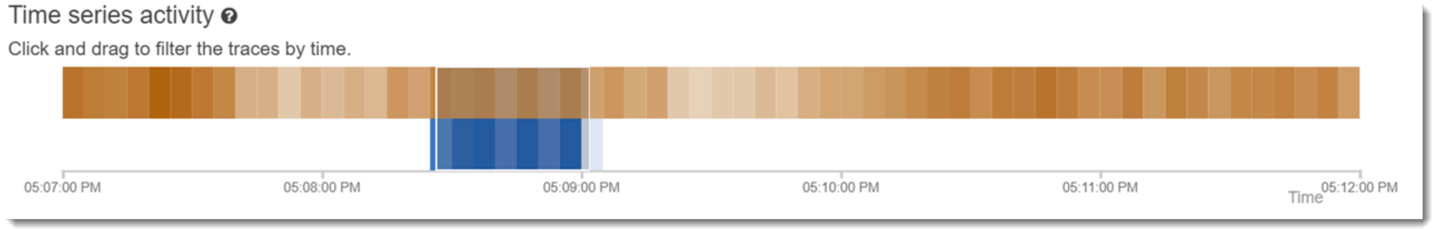
Click and drag to filter the traces by response time.

Time series activity

The time series activity chart shows the number of traces at a given time period. The color indicators mirror the line graph colors of the response time distribution. The darker and fuller the color block within the activity series, the more traces are represented at the given time.

Example – Time series activity

Click and drag to make selections within the time series activity graph. This selects and creates a filter named `timerange` on the working trace set for all traces within a specific range of time.



Time series activity

Click and drag to filter the traces by time.

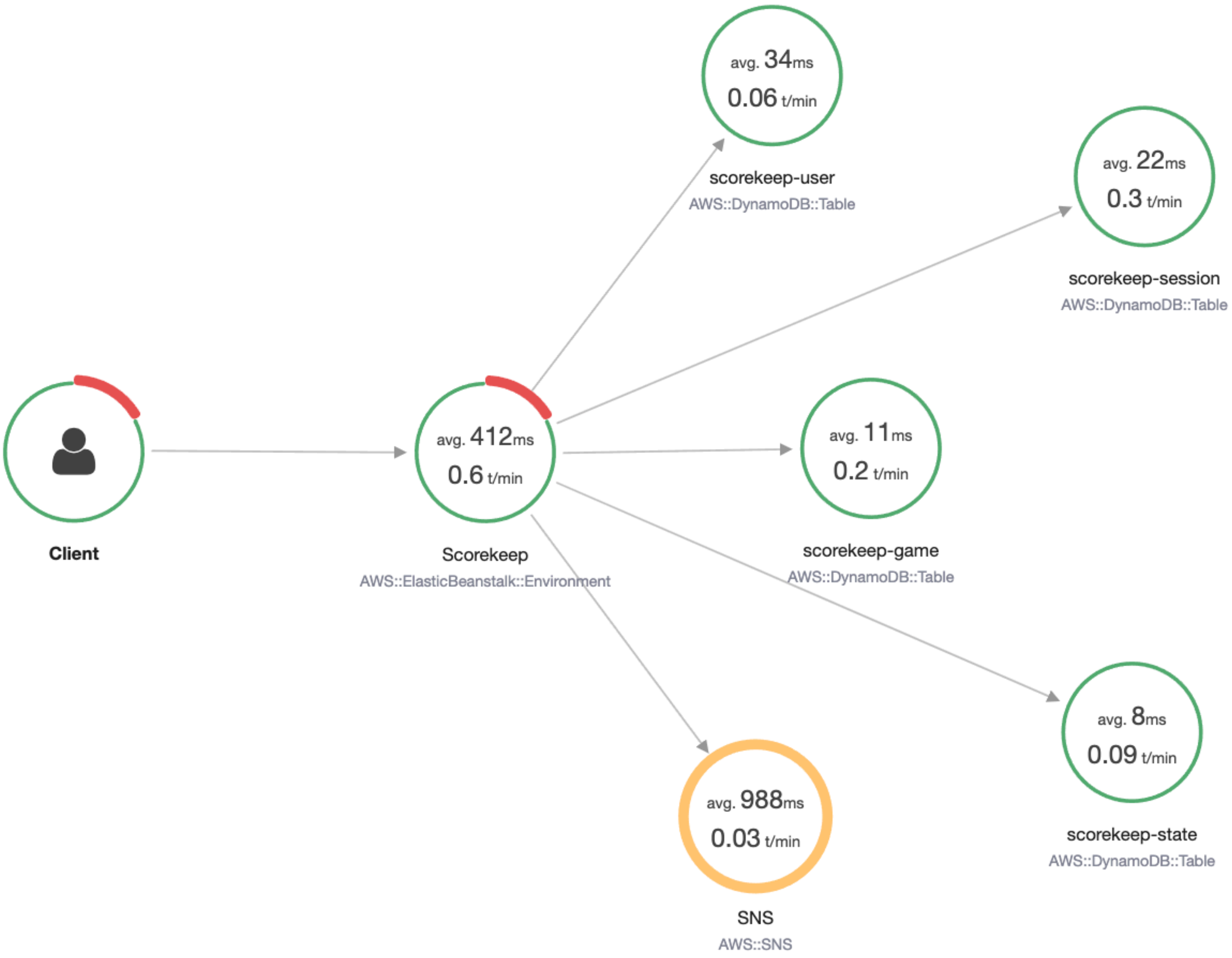
Workflow examples

The following examples show common use cases for the X-Ray Analytics console. Each example demonstrates a key function of the console experience. As a group, the examples follow a basic troubleshooting workflow. The steps walk through how to first spot unhealthy nodes, and then how to interact with the Analytics console to automatically generate comparative queries. Once you have narrowed the scope through queries, you will finally look at the details of traces of interest to determine what is damaging the health of your service.

Observe faults on the service graph

The trace map indicates the health of each node by coloring it based on the ratio of successful calls to errors and faults. When you see a percentage of red on your node, it signals a fault. Use the X-Ray Analytics console to investigate it.

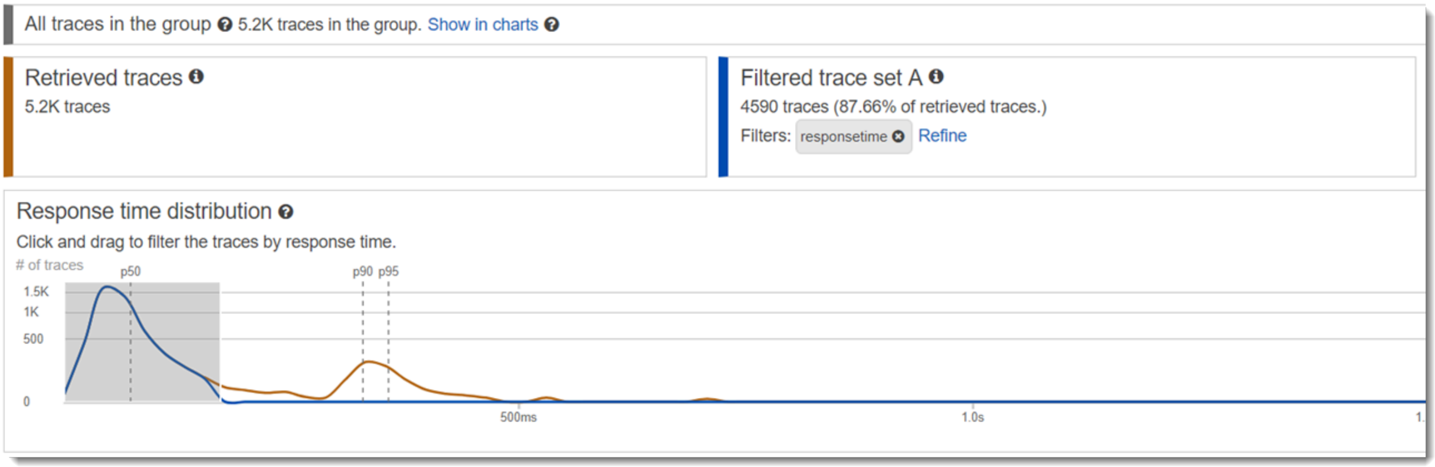
For more information about how to read the trace map, see Viewing the trace map.



Identify response time peaks

Using the response time distribution, you can observe peaks in response time. By selecting the peak in response time, the tables below the graphs will update to expose all associated metrics, such as status codes.

When you click and drag, X-Ray selects and creates a filter. It's shown in a gray shadow on top of the graphed lines. You can now drag that shadow left and right along the distribution to update your selection and filter.



All traces in the group 5.2K traces in the group. Show in charts

Retrieved traces

5.2K traces

Filtered trace set A

4590 traces (87.66% of retrieved traces.)

Filters: responsetime Refine

Response time distribution

Click and drag to filter the traces by response time.

View all traces marked with a status code

You can drill into traces within the selected peak by using the metrics tables below the graphs. By clicking a row in the **HTTP STATUS CODE** table, you automatically create a filter on the working dataset. For example, you could view all traces of status code 500. This creates a filter tag in the trace set tile named `http.status`.

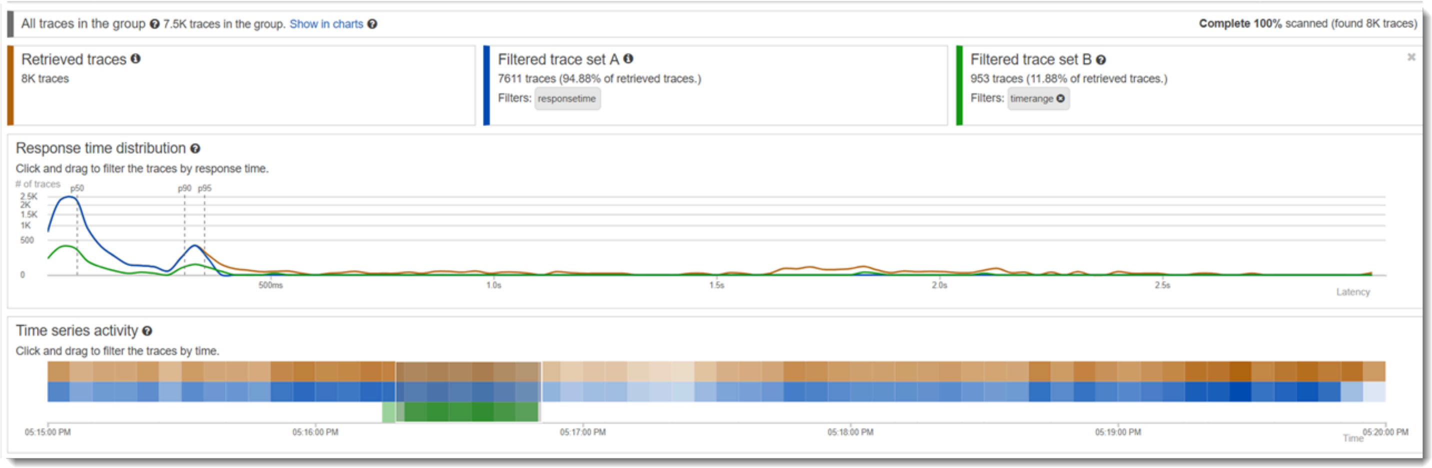
View all items in a subgroup and associated to a user

Drill into the error set based on user, URL, response time root cause, or other predefined attributes. For example, to additionally filter the set of traces with a 500 status code, select a row from the **USERS** table. This results in two filter tags in the trace set tile: `http.status`, as designated previously, and `user`.

Compare two sets of traces with different criteria

Compare across various users and their POST requests to find other discrepancies and correlations. Apply your first set of filters. They are defined by a blue line in the response time distribution. Then select **Compare**. Initially, this creates a copy of the filters on trace set A.

To proceed, define a new set of filters to apply to trace set B. This second set is represented by a green line. The following example shows different lines according to the blue and green color scheme.



All traces in the group 7.5K traces in the group. Show in charts

Complete 100% scanned (found 8K traces)

Retrieved traces
8K traces

Filtered trace set A
7611 traces (94.88% of retrieved traces.)
Filters: responsetime

Filtered trace set B
953 traces (11.88% of retrieved traces.)
Filters: timerange

Response time distribution

Click and drag to filter the traces by response time.

Time series activity

Click and drag to filter the traces by time.

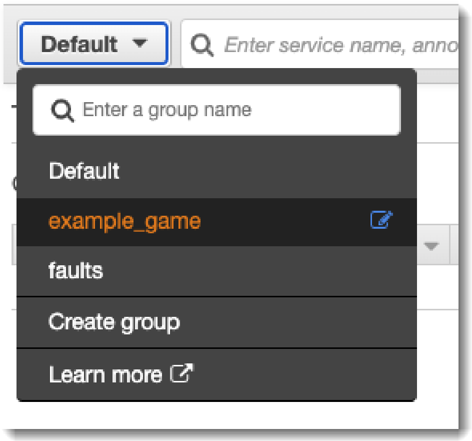
Identify a trace of interest and view its details

As you narrow your scope using the console filters, the trace list below the metrics tables becomes more meaningful. The trace list table combines information about **URL**, **USER**, and **STATUS CODE** into one view. For more insights, select a row from this table to open the trace's detail page and view its timeline and raw data.

Configuring groups

Groups are a collection of traces that are defined by a filter expression. You can use groups to generate additional service graphs and supply Amazon CloudWatch metrics. You can use the AWS X-Ray console or X-Ray API to create and manage groups for your services. This topic describes how to create and manage groups by using the X-Ray console. For information about how to manage groups by using the X-Ray API, see Groups.

You can create groups of traces for trace maps, traces, or analytics. When you create a group, the group becomes available as a filter on the group dropdown menu on all three pages: **Trace Map**, **Traces**, and **Analytics**.



Groups are identified by their name or an Amazon Resource Name (ARN), and contain a filter expression. The service compares incoming traces to the expression and stores them accordingly. For more information about how to build a filter expression, see Using filter expressions.

Updating a group's filter expression doesn't change data that's already recorded. The update applies only to subsequent traces. This can result in a merged graph of the new and old expressions. To avoid this, delete a current group and create a new one.

> **Note**
>
> Groups are billed by the number of retrieved traces that match the filter expression. For more information, see AWS X-Ray pricing.

Topics

- Create a group
- Apply a group
- Edit a group
- Clone a group
- Delete a group
- View group metrics in Amazon CloudWatch

Create a group

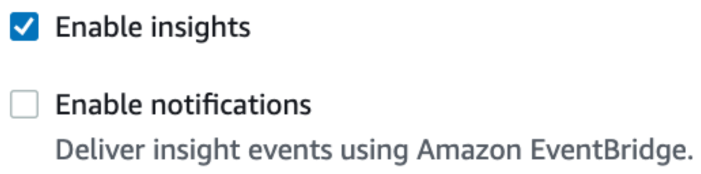
> **Note**
>
> You can now configure X-Ray groups from within the Amazon CloudWatch console. You can also continue to use the X-Ray console.

CloudWatch console

1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Choose **Settings** in the left navigation pane.
3. Choose **View settings** under **Groups** within the **X-Ray traces** section.
4. Choose **Create group** above the list of groups.
5. On the **Create group** page, enter a name for the group. A group name can have a maximum of 32 characters, and contain alphanumeric characters and dashes. Group names are case sensitive.
6. Enter a filter expression. For more information about how to build a filter expression, see Using filter expressions. In the following example, the group filters for fault traces from the service `api.example.com`. and requests to the service where the response time was greater than or equal to five seconds.

```
fault = true AND http.url CONTAINS "example/game" AND responsetime >= 5
```

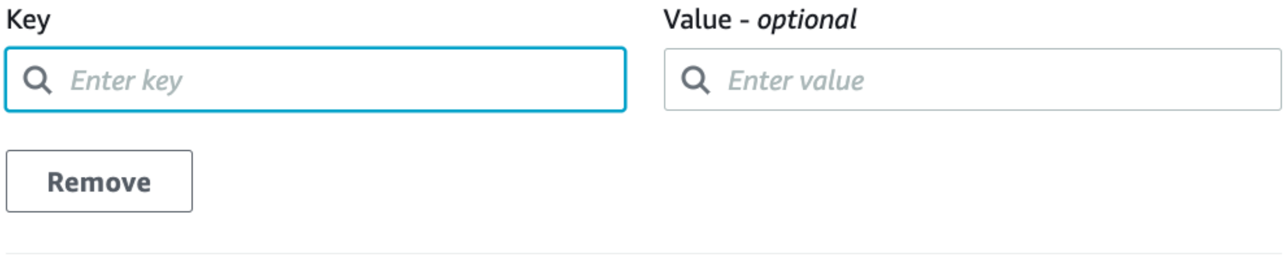
7. In **Insights**, enable or disable insights access for the group. For more information about insights, see Using X-Ray insights.



☑ Enable insights

□ Enable notifications
Deliver insight events using Amazon EventBridge.

8. In **Tags**, choose **Add new tag** to enter a tag key, and optionally, a tag value. Continue to add additional tags as desired. Tag keys must be unique. To delete a tag, choose **Remove** underneath each tag. For more information about tags, see Tagging X-Ray sampling rules and groups.



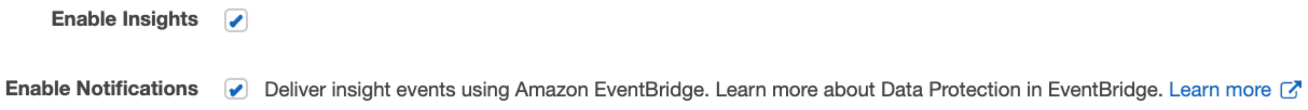
9. Choose **Create group**.

X-Ray console

1. Sign in to the AWS Management Console and open the X-Ray console at https://console.aws.amazon.com/xray/home.
2. Open the **Create group** page from the **Groups** page in the left navigation pane, or from the group menu on one of the following pages: **Trace Map**, **Traces**, and **Analytics**.
3. On the **Create group** page, enter a name for the group. A group name can have a maximum of 32 characters, and contain alphanumeric characters and dashes. Group names are case sensitive.
4. Enter a filter expression. For more information about how to build a filter expression, see [Using filter expressions](). In the following example, the group filters for fault traces from the service `api.example.com`. and requests to the service where the response time was greater than or equal to five seconds.

```
fault = true AND http.url CONTAINS "example/game" AND responsetime >= 5
```

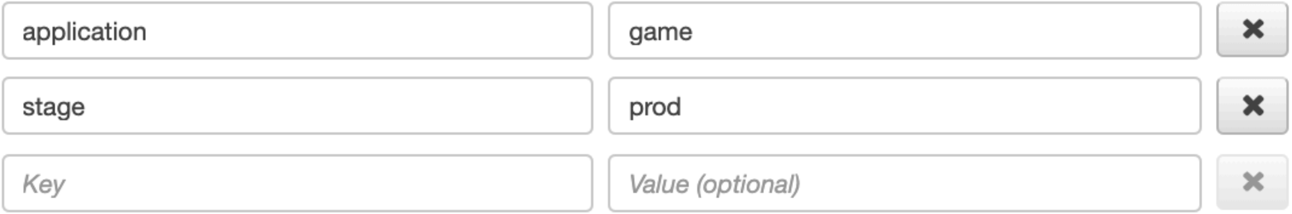
5. In **Insights**, enable or disable insights access for the group. For more information about insights, see [Using X-Ray insights]().



Enable Insights ☑

Enable Notifications ☑ Deliver insight events using Amazon EventBridge. Learn more about Data Protection in EventBridge. Learn more

6. In **Tags**, enter a tag key, and optionally, a tag value. As you add a tag, a new line appears for you to enter another tag. Tag keys must be unique. To delete a tag, choose **X** at the end of the tag's row. For more information about tags, see [Tagging X-Ray sampling rules and groups]().



application	game
stage	prod
Key	Value (optional)

7. Choose **Create group**.

Apply a group

CloudWatch console

1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Open one of the following pages from the navigation pane under **X-Ray traces**:
   - **Trace Map**
   - **Traces**
3. Enter a group name into the **Filter by X-Ray group** filter. The data shown on the page changes to match the filter expression set in the group.

X-Ray console

1. Sign in to the AWS Management Console and open the X-Ray console at https://console.aws.amazon.com/xray/home.
2. Open one of the following pages from the navigation pane:
   - **Trace Map**
   - **Traces**
   - **Analytics**
3. On the group menu, choose the group that you created in the section called “Create a group”. The data shown on the page changes to match the filter expression set in the group.

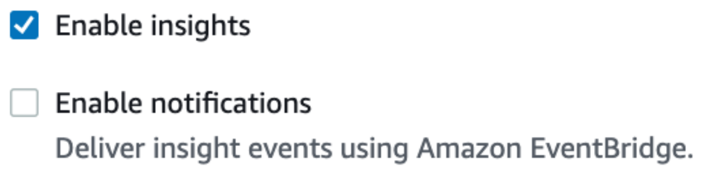
Edit a group

CloudWatch console

1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Choose **Settings** in the left navigation pane.
3. Choose **View settings** under **Groups** within the **X-Ray traces** section.
4. Choose a group from the **Groups** section and then choose **Edit**.
5. Although you can't rename a group, you can update the filter expression. For more information about how to build a filter expression, see Using filter expressions. In the following example, the group filters for fault traces from the service `api.example.com`, where the request URL address contains `example/game`, and response time for requests was greater than or equal to five seconds.

```
fault = true AND http.url CONTAINS "example/game" AND responsetime >= 5
```

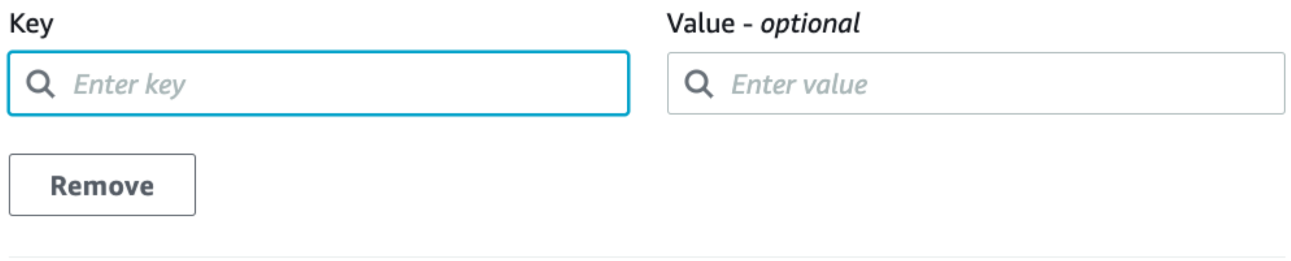
6. In **Insights**, enable or disable insights access for the group. For more information about insights, see Using X-Ray insights.



☑ Enable insights

□ Enable notifications
Deliver insight events using Amazon EventBridge.

7. In **Tags**, choose **Add new tag** to enter a tag key, and optionally, a tag value. Continue to add additional tags as desired. Tag keys must be unique. To delete a tag, choose **Remove** underneath each tag. For more information about tags, see Tagging X-Ray sampling rules and groups.



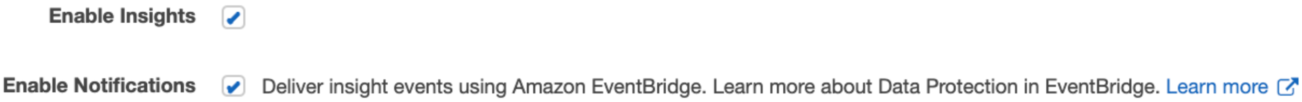
8. When you're finished updating the group, choose **Update group**.

X-Ray console

1. Sign in to the AWS Management Console and open the X-Ray console at https://console.aws.amazon.com/xray/home.
2. Do one of the following to open the **Edit group** page.
   a. On the **Groups** page, choose the name of a group to edit it.
   b. On the group menu on one of the following pages, point to a group, and then choose **Edit**.
      - **Trace Map**
      - **Traces**
      - **Analytics**
3. Although you can't rename a group, you can update the filter expression. For more information about how to build a filter expression, see Using filter expressions. In the following example, the group filters for fault traces from the service `api.example.com`, where the request URL address contains `example/game`, and response time for requests was greater than or equal to five seconds.

```
fault = true AND http.url CONTAINS "example/game" AND responsetime >= 5
```

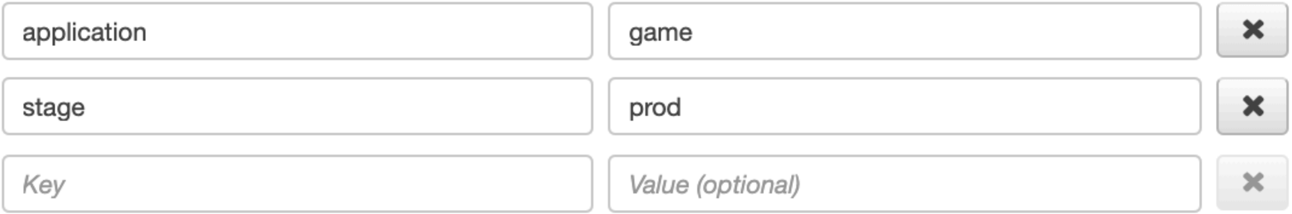
4. In **Insights**, enable or disable insights and insights notifications for the group. For more information about insights, see Using X-Ray insights.



Enable Insights ☑

Enable Notifications ☑ Deliver insight events using Amazon EventBridge. Learn more about Data Protection in EventBridge. Learn more

5. In **Tags**, edit tag keys and values. Tag keys must be unique. Tag values are optional; you can delete values, if you want. To delete a tag, choose **X** at the end of the tag's row. For more information about tags, see Tagging X-Ray sampling rules and groups.



application	game	✖
stage	prod	✖
Key	*Value (optional)*	✖

6. When you're finished updating the group, choose **Update group**.

Clone a group

Cloning a group creates a new group that has the filter expression and tags of an existing group. When you clone a group, the new group has the same name as the group from which it's cloned, with `-clone` appended to the name.

CloudWatch console

1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Choose **Settings** in the left navigation pane.
3. Choose **View settings** under **Groups** within the **X-Ray traces** section.
4. Choose a group from the **Groups** section and then choose **Clone**.
5. On the **Create group** page, the name of the group is *`group-name`*`-clone`. Optionally, enter a new name for the group. A group name can have a maximum of 32 characters, and contain alphanumeric characters and dashes. Group names are case sensitive.
6. You can keep the filter expression from the existing group, or optionally, enter a new filter expression. For more information about how to build a filter expression, see Using filter expressions. In the following example, the group filters for fault traces from the service `api.example.com`. and requests to the service where the response time was greater than or equal to five seconds.

   ```
   service("api.example.com") { fault = true OR responsetime >= 5 }
   ```

7. In **Tags**, edit tag keys and values, if needed. Tag keys must be unique. Tag values are optional; you can delete values if you want. To delete a tag, choose **X** at the end of the tag's row. For more information about tags, see Tagging X-Ray sampling rules and groups.
8. Choose **Create group**.

X-Ray console

1. Sign in to the AWS Management Console and open the X-Ray console at https://console.aws.amazon.com/xray/home.
2. Open the **Groups** page from the left navigation pane, and the choose the name of a group that you want to clone.
3. Choose **Clone group** from the **Actions** menu.

4. On the **Create group** page, the name of the group is `group-name-clone`. Optionally, enter a new name for the group. A group name can have a maximum of 32 characters, and contain alphanumeric characters and dashes. Group names are case sensitive.
5. You can keep the filter expression from the existing group, or optionally, enter a new filter expression. For more information about how to build a filter expression, see Using filter expressions. In the following example, the group filters for fault traces from the service `api.example.com`. and requests to the service where the response time was greater than or equal to five seconds.

```
service("api.example.com") { fault = true OR responsetime >= 5 }
```

6. In **Tags**, edit tag keys and values, if needed. Tag keys must be unique. Tag values are optional; you can delete values if you want. To delete a tag, choose **X** at the end of the tag's row. For more information about tags, see Tagging X-Ray sampling rules and groups.
7. Choose **Create group**.

Delete a group

Follow steps in this section to delete a group. You can't delete the **Default** group.

CloudWatch console

1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Choose **Settings** in the left navigation pane.
3. Choose **View settings** under **Groups** within the **X-Ray traces** section.
4. Choose a group from the **Groups** section and then choose **Delete**.
5. When you're prompted to confirm, choose **Delete**.

X-Ray console

1. Sign in to the AWS Management Console and open the X-Ray console at https://console.aws.amazon.com/xray/home.
2. Open the **Groups** page from the left navigation pane, and the choose the name of a group that you want to delete.
3. On the **Actions** menu, choose **Delete group**.

4. When you're prompted to confirm, choose **Delete**.

View group metrics in Amazon CloudWatch

After a group is created, incoming traces are checked against the group’s filter expression as they're stored in the X-Ray service. Metrics for the number of traces matching each criteria are published to Amazon CloudWatch every minute. Choosing **View metric** on the **Edit group** page opens the CloudWatch console to the **Metric** page. For more information about how to use CloudWatch metrics, see Using Amazon CloudWatch Metrics in the *Amazon CloudWatch User Guide*.

CloudWatch console

1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Choose **Settings** in the left navigation pane.
3. Choose **View settings** under **Groups** within the **X-Ray traces** section.
4. Choose a group from the **Groups** section and then choose **Edit**.
5. On the **Edit group** page, choose **View metric**.

   The CloudWatch console **Metrics** page opens in a new tab.

X-Ray console

1. Sign in to the AWS Management Console and open the X-Ray console at https://console.aws.amazon.com/xray/home.
2. Open the **Groups** page from the left navigation pane, and the choose the name of a group that you want to view metrics for.
3. On the **Edit group** page, choose **View metric**.

   The CloudWatch console **Metrics** page opens in a new tab.

Configuring sampling rules

You can use the AWS X-Ray console to configure sampling rules for your services. The X-Ray SDK and AWS services that support active tracing with sampling configuration use sampling rules to determine which requests to record.

Topics

- Configuring sampling rules
- Customizing sampling rules
- Sampling rule options
- Sampling rule examples
- Configuring your service to use sampling rules
- Viewing sampling results
- Next steps

Configuring sampling rules

You can configure sampling for the following use cases:

- **API Gateway Entrypoint** – API Gateway supports sampling and active tracing. To enable active tracing on an API stage, see Amazon API Gateway active tracing support for AWS X-Ray.
- **AWS AppSync** – AWS AppSync supports sampling and active tracing. To enable active tracing on AWS AppSync requests, see Tracing with AWS X-Ray.
- **Instrument X-Ray SDK on compute platforms** – When using compute platforms such as Amazon EC2, Amazon ECS, or AWS Elastic Beanstalk, sampling is supported when the application has been instrumented with the latest X-Ray SDK.

Customizing sampling rules

By customizing sampling rules, you can control the amount of data that you record. You can also modify sampling behavior without modifying or redeploying your code. Sampling rules tell the X-Ray SDK how many requests to record for a set of criteria. By default, the X-Ray SDK records the first request each second, and five percent of any additional requests. One request per second is the *reservoir*. This ensures that at least one trace is recorded each second as long as the service is serving requests. Five percent is the *rate* at which additional requests beyond the reservoir size are sampled.

You can configure the X-Ray SDK to read sampling rules from a JSON document that you include with your code. However, when you run multiple instances of your service, each instance performs sampling independently. This causes the overall percentage of requests sampled to increase because the reservoirs of all of the instances are effectively added together. Additionally, to update local sampling rules, you must redeploy your code.

By defining sampling rules in the X-Ray console, and configuring the SDK to read rules from the X-Ray service, you can avoid both of these issues. The service manages the reservoir for each rule, and assigns quotas to each instance of your service to distribute the reservoir evenly, based on the number of instances that are running. The reservoir limit is calculated according to the rules you set. Because the rules are configured in the service, you can manage rules without making additional deployments.

> **Note**
>
> X-Ray uses a best-effort approach in applying sampling rules, and in some cases the effective sampling rate may not exactly match the configured sampling rules. However, over time the number of requests sampled should be close to the configured percentage.

You can now configure X-Ray sampling rules from within the Amazon CloudWatch console. You can also continue to use the X-Ray console.

CloudWatch console

To configure sampling rules in the CloudWatch console

1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Choose **Settings** in the left navigation pane.
3. Choose **View settings** under **Sampling rules** within the **X-Ray traces** section.
4. To create a rule, choose **Create sampling rule**.

   To edit a rule, choose a rule and choose **Edit** to edit it.

   To delete a rule, choose a rule and choose **Delete** to delete it.

X-Ray console

To configure sampling rules in the X-Ray console

1. Open the X-Ray console.

2. Choose **Sampling** in the left navigation pane.
3. To create a rule, choose **Create sampling rule**.

   To edit a rule, choose a rule's name.

   To delete a rule, choose a rule and use the **Actions** menu to delete it.

Sampling rule options

The following options are available for each rule. String values can use wildcards to match a single character (?) or zero or more characters (*).

Sampling rule options

- **Rule name** (string) – A unique name for the rule.
- **Priority** (integer between 1 and 9999) – The priority of the sampling rule. Services evaluate rules in ascending order of priority, and make a sampling decision with the first rule that matches.
- **Reservoir** (non-negative integer) – A fixed number of matching requests to instrument per second, before applying the fixed rate. The reservoir is not used directly by services, but applies to all services using the rule collectively.
- **Rate** (integer between 0 and 100) – The percentage of matching requests to instrument, after the reservoir is exhausted. When configuring a sampling rule in the console, choose a percentage between 0 and 100. When configuring a sampling rule in a client SDK using a JSON document, provide a percentage value between 0 and 1.
- **Service name** (string) – The name of the instrumented service, as it appears in the trace map.
  - X-Ray SDK – The service name that you configure on the recorder.
  - Amazon API Gateway – *`api-name`*`/`*`stage`*.
- **Service type** (string) – The service type, as it appears in the trace map. For the X-Ray SDK, set the service type by applying the appropriate plugin:
  - `AWS::ElasticBeanstalk::Environment` – An AWS Elastic Beanstalk environment (plugin).
  - `AWS::EC2::Instance` – An Amazon EC2 instance (plugin).
  - `AWS::ECS::Container` – An Amazon ECS container (plugin).
  - `AWS::APIGateway::Stage` – An Amazon API Gateway stage.
  - `AWS::AppSync::GraphQLAPI` – An AWS AppSync API request.
- **Host** (string) – The hostname from the HTTP host header.

- **HTTP method** (string) – The method of the HTTP request.
- **URL path** (string) – The URL path of the request.
  - X-Ray SDK – The path portion of the HTTP request URL.
- **Resource ARN** (string) – The ARN of the AWS resource running the service.
  - X-Ray SDK – Not supported. The SDK can only use rules with **Resource ARN** set to `*`.
  - Amazon API Gateway – The stage ARN.
- (Optional) **Attributes** (key and value) – Segment attributes that are known when the sampling decision is made.
  - X-Ray SDK – Not supported. The SDK ignores rules that specify attributes.
  - Amazon API Gateway – Headers from the original HTTP request.

Sampling rule examples

Example – Default rule with no reservoir and a low rate

You can modify the default rule's reservoir and rate. The default rule applies to requests that don't match any other rule.

- **Reservoir**: **`0`**
- **Rate**: **5** (**`0.05`** if configured using a JSON document)

Example – Debugging rule to trace all requests for a problematic route

A high-priority rule applied temporarily for debugging.

- **Rule name**: **`DEBUG - history updates`**
- **Priority**: **1**
- **Reservoir**: **1**
- **Rate**: **`100`** (**1** if configured using a JSON document)
- **Service name**: **`Scorekeep`**
- **Service type**: **`*`**
- **Host**: **`*`**
- **HTTP method**: **PUT**
- **URL path**: **`/history/*`**

- **Resource ARN**: *****

Example – Higher minimum rate for POSTs

- **Rule name**: **`POST minimum`**
- **Priority**: **`100`**
- **Reservoir**: **`10`**
- **Rate**: **`10`** (**`.1`** if configured using a JSON document)
- **Service name**: *****
- **Service type**: *****
- **Host**: *****
- **HTTP method**: **POST**
- **URL path**: *****
- **Resource ARN**: *****

Configuring your service to use sampling rules

The X-Ray SDK requires additional configuration to use sampling rules that you configure in the console. See the configuration topic for your language for details on configuring a sampling strategy:

- Java: Sampling rules
- Go: Sampling rules
- Node.js: Sampling rules
- Python: Sampling rules
- Ruby: Sampling rules
- .NET: Sampling rules

For API Gateway, see Amazon API Gateway active tracing support for AWS X-Ray.

Viewing sampling results

The X-Ray console **Sampling** page shows detailed information about how your services use each sampling rule.

The **Trend** column shows how the rule has been used in the last few minutes. Each column shows statistics for a 10-second window.

Sampling statistics

- **Total matched rule**: The number of requests that matched this rule. This number doesn't include requests that could have matched this rule, but matched a higher-priority rule first.
- **Total sampled**: The number of requests recorded.
- **Sampled with fixed rate**: The number of requests sampled by applying the rule's fixed rate.
- **Sampled with reservoir limit**: The number of requests sampled using a quota assigned by X-Ray.
- **Borrowed from reservoir**: The number of requests sampled by borrowing from the reservoir. The first time a service matches a request to a rule, it has not yet been assigned a quota by X-Ray. However, if the reservoir is at least 1, the service borrows one trace per second until X-Ray assigns a quota.

For more information about sampling statistics and how services use sampling rules, see Using sampling rules with the X-Ray API.

Next steps

You can use the X-Ray API to manage sampling rules. With the API, you can create and update rules programmatically on a schedule, or in response to alarms or notifications. See Configuring sampling, groups, and encryption settings with the AWS X-Ray API for instructions and additional rule examples.

The X-Ray SDK and AWS services also use the X-Ray API to read sampling rules, report sampling results, and get sampling targets. Services must keep track of how often they apply each rule, evaluate rules based on priority, and borrow from the reservoir when a request matches a rule for which X-Ray has not yet assigned the service a quota. For more detail about how a service uses the API for sampling, see Using sampling rules with the X-Ray API.

When the X-Ray SDK calls sampling APIs, it uses the X-Ray daemon as a proxy. If you already use TCP port 2000, you can configure the daemon to run the proxy on a different port. See Configuring the AWS X-Ray daemon for details.

Console deep linking

You can use routes and queries to deep link into specific traces, or filtered views of traces and the trace map.

Console pages

- Welcome page – xray/home#/welcome
- Getting started – xray/home#/getting-started
- Trace map – xray/home#/service-map
- Traces – xray/home#/traces

Traces

You can generate links for timeline, raw, and map views of individual traces.

Trace timeline – `xray/home#/traces/`*`trace-id`*

Raw trace data – `xray/home#/traces/`*`trace-id`*`/raw`

Example – raw trace data

```
https://console.aws.amazon.com/xray/home#/traces/1-57f5498f-d91047849216d0f2ea3b6442/
raw
```

Filter expressions

Link to a filtered list of traces.

Filtered traces view – `xray/home#/traces?filter=`*`filter-expression`*

Example – filter expression

```
https://console.aws.amazon.com/xray/home#/traces?filter=service("api.amazon.com")
 { fault = true OR responsetime > 2.5 } AND annotation.foo = "bar"
```

Example – filter expression (URL encoded)

```
https://console.aws.amazon.com/xray/home#/traces?filter=service(%22api.amazon.com
%22)%20%7B%20fault%20%3D%20true%20OR%20responsetime%20%3E%202.5%20%7D%20AND
%20annotation.foo%20%3D%20%22bar%22
```

For more information about filter expressions, see Using filter expressions.

Time range

Specify a length of time or start and end time in ISO8601 format. Time ranges are in UTC and can be up to 6 hours long.

Length of time – `xray/home#/`*`page`*`?timeRange=`*`range-in-minutes`*

Example – trace map for the last hour

```
https://console.aws.amazon.com/xray/home#/service-map?timeRange=PT1H
```

Start and end time – `xray/home#/`*`page`*`?timeRange=`*`start`*`~`*`end`*

Example – time range accurate to seconds

```
https://console.aws.amazon.com/xray/home#/traces?
timeRange=2023-7-01T16:00:00~2023-7-01T22:00:00
```

Example – time range accurate to minutes

```
https://console.aws.amazon.com/xray/home#/traces?
timeRange=2023-7-01T16:00~2023-7-01T22:00
```

Region

Specify an AWS Region to link to pages in that Region. If you don't specify a Region, the console redirects you to the last visited Region.

Region – `xray/home`**`?region=`*`region`***`#/`*`page`*

Example – trace map in US West (Oregon) (us-west-2)

```
https://console.aws.amazon.com/xray/home?region=us-west-2#/service-map
```

When you include a Region with other query parameters, the Region query goes before the hash, and the X-Ray-specific queries go after the page name.

Example – trace map for the last hour in US West (Oregon) (us-west-2)

```
https://console.aws.amazon.com/xray/home?region=us-west-2#/service-map?timeRange=PT1H
```

Combined

Example – recent traces with a duration filter

```
https://console.aws.amazon.com/xray/home#/traces?timeRange=PT15M&filter=duration%20%3E
%3D%205%20AND%20duration%20%3C%3D%208
```

Output

- Page – Traces
- Time Range – Last 15 minutes
- Filter – duration >= 5 AND duration <= 8

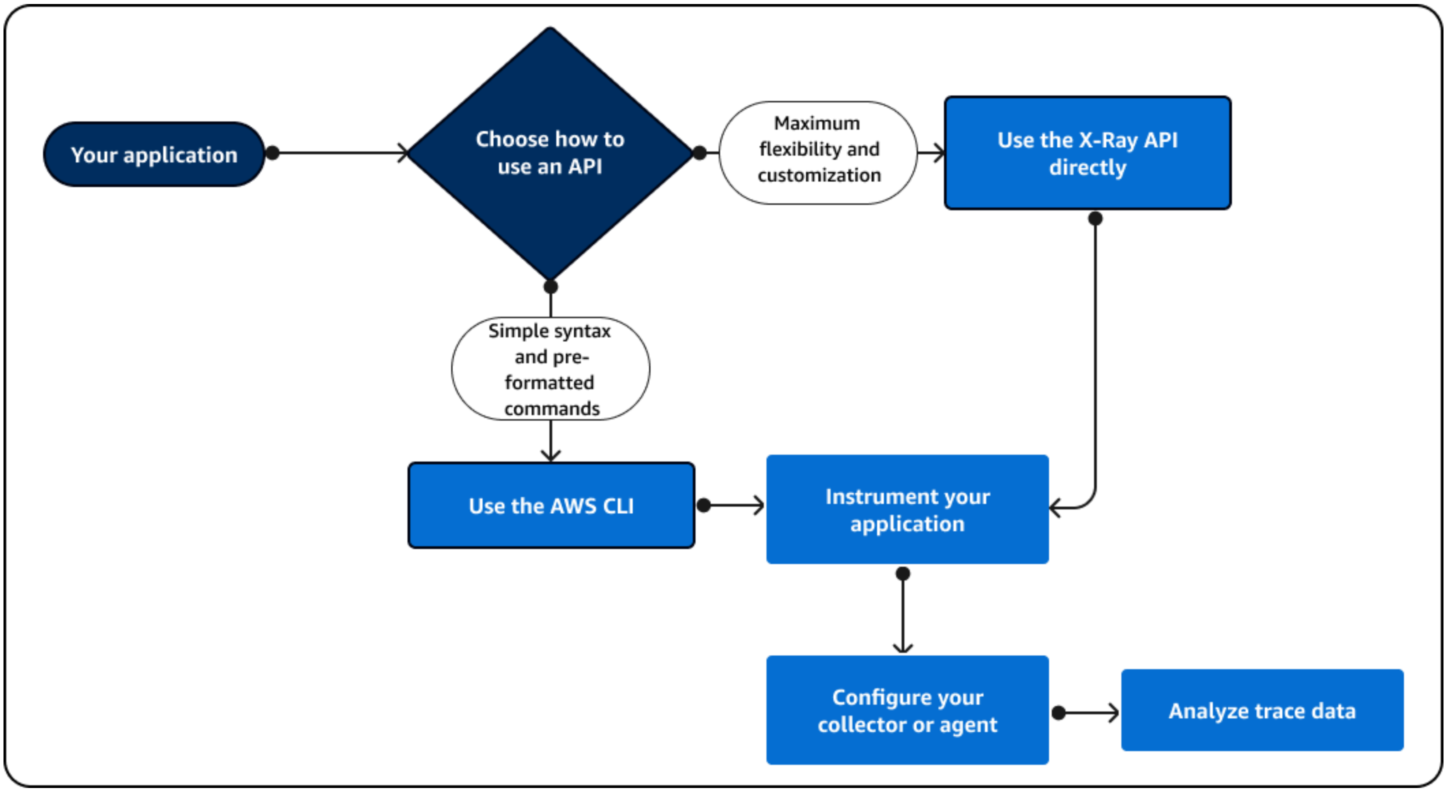
Use the X-Ray API

If the X-Ray SDK doesn't support your programming language, you can use either the X-Ray APIs directly or the AWS Command Line Interface (AWS CLI) to call X-Ray API commands. Use the following guidance to choose how you interact with the API:

- Use the AWS CLI for simpler syntax using pre-formatted commands or with options inside your request.
- Use the X-Ray API directly for maximum flexibility and customization for requests that you make to X-Ray.

If you use the X-Ray API directly instead of the AWS CLI, you must parametrize your request in the correct data format and may also have to configure authentication and error handling.

The following diagram shows guidance to choose how to interact with the X-Ray API:


Your application
Choose how to use an API
Maximum flexibility and customization
Use the X-Ray API directly
Simple syntax and pre-formatted commands
Use the AWS CLI
Instrument your application
Configure your collector or agent
Analyze trace data

Use the X-Ray API to send trace data to directly to X-Ray. The X-Ray API supports all functions available in the X-Ray SDK including the following common actions:

- PutTraceSegments – Uploads segment documents to X-Ray.
- BatchGetTraces – Retrieves a list of traces in a list of trace IDs. Each retrieved trace is a collection of segment documents from a single request.
- GetTraceSummaries – Retrieves IDs and annotations for traces. You can specify a `FilterExpression` to retrieve a subset of trace summaries.
- GetTraceGraph – Retrieves a service graph for a specific trace ID.
- GetServiceGraph – Retrieves a JSON formatted document that describes services that process incoming requests and call downstream requests.

You can also use the AWS Command Line Interface (AWS CLI) inside your application code to programmatically interact with X-Ray. The AWS CLI supports all functions available in the X-Ray SDK including those for other AWS services. The following functions are versions of the API operations listed previously with a simpler format:

- put-trace-segments – Uploads segment documents to X-Ray.

- batch-get-traces – Retrieves a list of traces in a list of trace IDs. Each retrieved trace is a collection of segment documents from a single request.
- get-trace-summaries – Retrieves IDs and annotations for traces. You can specify a `FilterExpression` to retrieve a subset of trace summaries.
- get-trace-graph – Retrieves a service graph for a specific trace ID.
- get-service-graph – Retrieves a `JSON` formatted document that describes services that process incoming requests and call downstream requests.

To get started, you must install the AWS CLI for your operating system. AWS supports Linux, macOS and Windows operating systems. For more information about the list of X-Ray commands, see the AWS CLI Command Reference guide for X-Ray.

Topics

- Using the AWS X-Ray API with the AWS CLI
- Sending trace data to AWS X-Ray
- Getting data from AWS X-Ray
- Configuring sampling, groups, and encryption settings with the AWS X-Ray API
- Using sampling rules with the X-Ray API
- AWS X-Ray segment documents

Using the AWS X-Ray API with the AWS CLI

The AWS CLI lets your access the X-Ray service directly and use the same APIs that the X-Ray console uses to retrieve the service graph and raw traces data. The sample application includes scripts that show how to use these APIs with the AWS CLI.

Prerequisites

This tutorial uses the Scorekeep sample application and included scripts to generate tracing data and a service map. Follow the instructions in the getting started tutorial to launch the application.

This tutorial uses the AWS CLI to show basic use of the X-Ray API. The AWS CLI, available for Windows, Linux, and OS-X, provides command line access to the public APIs for all AWS services.

> **Note**
>
> You must verify that your AWS CLI is configured to the same Region that your Scorekeep sample application was created in.

Scripts included to test the sample application uses cURL to send traffic to the API and jq to parse the output. You can download the jq executable from stedolan.github.io, and the curl executable from https://curl.haxx.se/download.html. Most Linux and OS X installations include cURL.

Generate trace data

The web app continues to generate traffic to the API every few seconds while the game is in-progress, but only generates one type of request. Use the test-api.sh script to run end to end scenarios and generate more diverse trace data while you test the API.

To use the test-api.sh script

1. Open the Elastic Beanstalk console.
2. Navigate to the management console for your environment.
3. Copy the environment **URL** from the page header.
4. Open bin/test-api.sh and replace the value for API with your environment's URL.

```
#!/bin/bash
API=scorekeep.9hbtbm23t2.us-west-2.elasticbeanstalk.com/api
```

5. Run the script to generate traffic to the API.

```
~/debugger-tutorial$ ./bin/test-api.sh
Creating users,
session,
game,
configuring game,
playing game,
ending game,
game complete.
{"id":"MTBP8BAS","session":"HUF6IT64","name":"tic-tac-toe-test","users":
["QFF3HBGM","KL6JR98D"],"rules":"102","startTime":1476314241,"endTime":1476314245,"states":
["JQVLEOM2","D67QLPIC","VF9BM9NC","OEAA6GK9","2A705073","1U2LFTLJ","HUKIDD70","BAN1C8FI","G
["BS8F8LQ","4MTTSPKP","4630ETES","SVEBCL3N","N7CQ1GHP","O840NEPD","EG4BPROQ","V4BLIDJ3","9R
```

Use the X-Ray API

The AWS CLI provides commands for all of the API actions that X-Ray provides, including GetServiceGraph and GetTraceSummaries. See the AWS X-Ray API Reference for more information on all of the supported actions and the data types that they use.

Example bin/service-graph.sh

```
EPOCH=$(date +%s)
aws xray get-service-graph --start-time $(($EPOCH-600)) --end-time $EPOCH
```

The script retrieves a service graph for the last 10 minutes.

```
~/eb-java-scorekeep$ ./bin/service-graph.sh | less
{
    "StartTime": 1479068648.0,
    "Services": [
        {
            "StartTime": 1479068648.0,
            "ReferenceId": 0,
            "State": "unknown",
            "EndTime": 1479068651.0,
            "Type": "client",
            "Edges": [
                {
                    "StartTime": 1479068648.0,
                    "ReferenceId": 1,
                    "SummaryStatistics": {
                        "ErrorStatistics": {
                            "ThrottleCount": 0,
                            "TotalCount": 0,
                            "OtherCount": 0
                        },
                        "FaultStatistics": {
                            "TotalCount": 0,
                            "OtherCount": 0
                        },
                        "TotalCount": 2,
                        "OkCount": 2,
                        "TotalResponseTime": 0.054000139236450195
                    },
                    "EndTime": 1479068651.0,
                    "Aliases": []
                }
            ]
        },
        {
            "StartTime": 1479068648.0,
            "Names": [
                "scorekeep.elasticbeanstalk.com"
            ],
            "ReferenceId": 1,
            "State": "active",
            "EndTime": 1479068651.0,
            "Root": true,
            "Name": "scorekeep.elasticbeanstalk.com",
...
```

Example bin/trace-urls.sh

```
EPOCH=$(date +%s)
aws xray get-trace-summaries --start-time $(($EPOCH-120)) --end-time $(($EPOCH-60)) --
query 'TraceSummaries[*].Http.HttpURL'
```

The script retrieves the URLs of traces generated between one and two minutes ago.

```
~/eb-java-scorekeep$ ./bin/trace-urls.sh
[
    "http://scorekeep.elasticbeanstalk.com/api/game/6Q0UE1DG/5FGLM9U3/
endtime/1479069438",
    "http://scorekeep.elasticbeanstalk.com/api/session/KH4341QH",
    "http://scorekeep.elasticbeanstalk.com/api/game/GLQBJ3K5/153AHDIA",
    "http://scorekeep.elasticbeanstalk.com/api/game/VPDL672J/G2V41HM6/
endtime/1479069466"
]
```

Example bin/full-traces.sh

```
EPOCH=$(date +%s)
TRACEIDS=$(aws xray get-trace-summaries --start-time $(($EPOCH-120)) --end-time
 $(($EPOCH-60)) --query 'TraceSummaries[*].Id' --output text)
aws xray batch-get-traces --trace-ids $TRACEIDS --query 'Traces[*]'
```

The script retrieves full traces generated between one and two minutes ago.

```
~/eb-java-scorekeep$ ./bin/full-traces.sh | less
[
    {
        "Segments": [
            {
                "Id": "3f212bc237bafd5d",
                "Document": "{\"id\":\"3f212bc237bafd5d\",\"name\":\"DynamoDB\",
\"trace_id\":\"1-5828d9f2-a90669393f4343211bc1cf75\",\"start_time\":1.479072242459E9,
\"end_time\":1.479072242477E9,\"parent_id\":\"72a08dcf87991ca9\",\"http\":
{\"response\":{\"content_length\":60,\"status\":200}},\"inferred\":true,\"aws\":
{\"consistent_read\":false,\"table_name\":\"scorekeep-session-xray\",\"operation\":
\"GetItem\",\"request_id\":\"QAKE0S8DD0LJM245KAOPMA746BVV4KQNSO5AEMVJF66Q9ASUAAJG\",
\"resource_names\":[\"scorekeep-session-xray\"]},\"origin\":\"AWS::DynamoDB::Table\"}"
            },
            {
                "Id": "309e355f1148347f",
                "Document": "{\"id\":\"309e355f1148347f\",\"name\":\"DynamoDB\",
\"trace_id\":\"1-5828d9f2-a90669393f4343211bc1cf75\",\"start_time\":1.479072242477E9,
\"end_time\":1.479072242494E9,\"parent_id\":\"37f14ef837f00022\",\"http\":
{\"response\":{\"content_length\":606,\"status\":200}},\"inferred\":true,\"aws\":
{\"table_name\":\"scorekeep-game-xray\",\"operation\":\"UpdateItem\",\"request_id
\":\"388GEROC4PCA6D59ED3CTI5EEJVV4KQNSO5AEMVJF66Q9ASUAAJG\",\"resource_names\":
[\"scorekeep-game-xray\"]},\"origin\":\"AWS::DynamoDB::Table\"}"
            }
        ],
        "Id": "1-5828d9f2-a90669393f4343211bc1cf75",
        "Duration": 0.050999879837036133
    }
...
```

Cleanup

Terminate your Elastic Beanstalk environment to shut down the Amazon EC2 instances, DynamoDB tables and other resources.

To terminate your Elastic Beanstalk environment

1. Open the Elastic Beanstalk console.
2. Navigate to the management console for your environment.
3. Choose **Actions**.
4. Choose **Terminate Environment**.

5. Choose **Terminate**.

Trace data is automatically deleted from X-Ray after 30 days.

Sending trace data to AWS X-Ray

You can send trace data to X-Ray in the form of segment documents. A segment document is a JSON formatted string that contains information about the work that your application does in service of a request. Your application can record data about the work that it does itself in segments, or work that uses downstream services and resources in subsegments.

Segments record information about the work that your application does. A segment, at a minimum, records the time spent on a task, a name, and two IDs. The trace ID tracks the request as it travels between services. The segment ID tracks the work done for the request by a single service.

Example Minimal complete segment

```
{
  "name" : "Scorekeep",
  "id" : "70de5b6f19ff9a0a",
  "start_time" : 1.478293361271E9,
  "trace_id" : "1-581cf771-a006649127e371903a2de979",
  "end_time" : 1.478293361449E9
}
```

When a request is received, you can send an in-progress segment as a placeholder until the request is completed.

Example In-progress segment

```
{
  "name" : "Scorekeep",
  "id" : "70de5b6f19ff9a0b",
  "start_time" : 1.478293361271E9,
  "trace_id" : "1-581cf771-a006649127e371903a2de979",
  “in_progress”: true
}
```

You can send segments to X-Ray directly, with `PutTraceSegments`, or through the X-Ray daemon.

Most applications call other services or access resources with the AWS SDK. Record information about downstream calls in *subsegments*. X-Ray uses subsegments to identify downstream services that don't send segments and create entries for them on the service graph.

A subsegment can be embedded in a full segment document, or sent separately. Send subsegments separately to asynchronously trace downstream calls for long-running requests, or to avoid exceeding the maximum segment document size (64 kB).

Example Subsegment

A subsegment has a `type` of `subsegment` and a `parent_id` that identifies the parent segment.

```
{
  "name" : "www2.example.com",
  "id" : "70de5b6f19ff9a0c",
  "start_time" : 1.478293361271E9,
  "trace_id" : "1-581cf771-a006649127e371903a2de979"
  “end_time” : 1.478293361449E9,
  “type” : “subsegment”,
  “parent_id” : “70de5b6f19ff9a0b”
}
```

For more information on the fields and values that you can include in segments and subsegments, see AWS X-Ray segment documents.

Sections

- Generating trace IDs
- Using PutTraceSegments
- Sending segment documents to the X-Ray daemon

Generating trace IDs

To send data to X-Ray, you must generate a unique trace ID for each request.

X-Ray trace ID format

An X-Ray `trace_id` consists of three numbers separated by hyphens. For example, `1-58406520-a006649127e371903a2de979`. This includes:

- The version number, which is 1.

- The time of the original request in Unix epoch time using **8 hexadecimal digits**.

  For example, 10:00AM December 1st, 2016 PST in epoch time is `1480615200` seconds or `58406520` in hexadecimal digits.
- A globally unique 96-bit identifier for the trace in **24 hexadecimal digits**.

> **Note**
>
> X-Ray now supports trace IDs that are created using OpenTelemetry and any other framework that conforms with the [W3C Trace Context specification](). A W3C trace ID must be formatted in X-Ray trace ID format when sending to X-Ray. For example, W3C trace ID `4efaaf4d1e8720b39541901950019ee5` should be formatted as `1-4efaaf4d-1e8720b39541901950019ee5` when sending to X-Ray. X-Ray trace IDs include the original request time stamp in Unix epoch time, but this isn't required when sending W3C trace IDs in X-Ray format.

You can write a script to generate X-Ray trace IDs for testing. Here are two examples.

Python

```
import time
import os
import binascii

START_TIME = time.time()
HEX=hex(int(START_TIME))[2:]
TRACE_ID="1-{}-{}".format(HEX, binascii.hexlify(os.urandom(12)).decode('utf-8'))
```

Bash

```
START_TIME=$(date +%s)
HEX_TIME=$(printf '%x\n' $START_TIME)
GUID=$(dd if=/dev/random bs=12 count=1 2>/dev/null | od -An -tx1 | tr -d ' \t\n')
TRACE_ID="1-$HEX_TIME-$GUID"
```

See the Scorekeep sample application for scripts that create trace IDs and send segments to the X-Ray daemon.

- Python – xray_start.py
- Bash – xray_start.sh

Using PutTraceSegments

You can upload segment documents with the PutTraceSegments API. The API has a single parameter, TraceSegmentDocuments, that takes a list of JSON segment documents.

With the AWS CLI, use the aws xray put-trace-segments command to send segment documents directly to X-Ray.

```
$ DOC='{"trace_id": "1-5960082b-ab52431b496add878434aa25", "id": "6226467e3f845502",
 "start_time": 1498082657.37518, "end_time": 1498082695.4042, "name":
 "test.elasticbeanstalk.com"}'
$ aws xray put-trace-segments --trace-segment-documents "$DOC"
{
    "UnprocessedTraceSegments": []
}
```

Note

Windows Command Processor and Windows PowerShell have different requirements for quoting and escaping quotes in JSON strings. See Quoting Strings in the AWS CLI User Guide for details.

The output lists any segments that failed processing. For example, if the date in the trace ID is too far in the past, you see an error like the following.

```
{
    "UnprocessedTraceSegments": [
        {
            "ErrorCode": "InvalidTraceId",
            "Message": "Invalid segment. ErrorCode: InvalidTraceId",
            "Id": "6226467e3f845502"
        }
    ]
}
```

You can pass multiple segment documents at the same time, separated by spaces.

```
$ aws xray put-trace-segments --trace-segment-documents "$DOC1" "$DOC2"
```

Sending segment documents to the X-Ray daemon

Instead of sending segment documents to the X-Ray API, you can send segments and subsegments to the X-Ray daemon, which will buffer them and upload to the X-Ray API in batches. The X-Ray SDK sends segment documents to the daemon to avoid making calls to AWS directly.

Note

See Running the X-Ray daemon locally for instructions on running the daemon.

Send the segment in JSON over UDP port 2000, prepended by the daemon header, `{"format": "json", "version": 1}\n`

```
{"format": "json", "version": 1}\n{"trace_id": "1-5759e988-bd862e3fe1be46a994272793",
 "id": "defdfd9912dc5a56", "start_time": 1461096053.37518, "end_time": 1461096053.4042,
 "name": "test.elasticbeanstalk.com"}
```

On Linux, you can send segment documents to the daemon from a Bash terminal. Save the header and segment document to a text file and pipe it to `/dev/udp` with `cat`.

```
$ cat segment.txt > /dev/udp/127.0.0.1/2000
```

Example segment.txt

```
{"format": "json", "version": 1}
{"trace_id": "1-594aed87-ad72e26896b3f9d3a27054bb", "id": "6226467e3f845502",
 "start_time": 1498082657.37518, "end_time": 1498082695.4042, "name":
 "test.elasticbeanstalk.com"}
```

Check the daemon log to verify that it sent the segment to X-Ray.

```
2017-07-07T01:57:24Z [Debug] processor: sending partial batch
2017-07-07T01:57:24Z [Debug] processor: segment batch size: 1. capacity: 50
2017-07-07T01:57:24Z [Info] Successfully sent batch of 1 segments (0.020 seconds)
```

Getting data from AWS X-Ray

AWS X-Ray processes the trace data that you send to it to generate full traces, trace summaries, and service graphs in JSON. You can retrieve the generated data directly from the API with the AWS CLI.

Sections

- Retrieving the service graph
- Retrieving the service graph by group
- Retrieving traces
- Retrieving and refining root cause analytics

Retrieving the service graph

You can use the `GetServiceGraph` API to retrieve the JSON service graph. The API requires a start time and end time, which you can calculate from a Linux terminal with the `date` command.

```
$ date +%s
1499394617
```

`date +%s` prints a date in seconds. Use this number as an end time and subtract time from it to get a start time.

Example Script to retrieve a service graph for the last 10 minutes

```
EPOCH=$(date +%s)
aws xray get-service-graph --start-time $(($EPOCH-600)) --end-time $EPOCH
```

The following example shows a service graph with 4 nodes, including a client node, an EC2 instance, a DynamoDB table, and an Amazon SNS topic.

Example GetServiceGraph output

```
{
    "Services": [
        {
            "ReferenceId": 0,
            "Name": "xray-sample.elasticbeanstalk.com",
            "Names": [
                "xray-sample.elasticbeanstalk.com"
            ],
            "Type": "client",
            "State": "unknown",
            "StartTime": 1528317567.0,
            "EndTime": 1528317589.0,
            "Edges": [
                {
                    "ReferenceId": 2,
                    "StartTime": 1528317567.0,
                    "EndTime": 1528317589.0,
                    "SummaryStatistics": {
                        "OkCount": 3,
                        "ErrorStatistics": {
                            "ThrottleCount": 0,
                            "OtherCount": 1,
                            "TotalCount": 1
                        },
                        "FaultStatistics": {
                            "OtherCount": 0,
                            "TotalCount": 0
                        },
                        "TotalCount": 4,
                        "TotalResponseTime": 0.273
                    },
                    "ResponseTimeHistogram": [
                        {
                            "Value": 0.005,
                            "Count": 1
                        },
                        {
                            "Value": 0.015,
                            "Count": 1
                        },
                        {
                            "Value": 0.157,
                            "Count": 1
                        },
                        {
                            "Value": 0.096,
                            "Count": 1
                        }
                    ],
                    "Aliases": []
                }
            ]
        },
        {
            "ReferenceId": 1,
            "Name": "awseb-e-dixzws4s9p-stack-StartupSignupsTable-4IMSMHAYX2BA",
            "Names": [
                "awseb-e-dixzws4s9p-stack-StartupSignupsTable-4IMSMHAYX2BA"
            ],
            "Type": "AWS::DynamoDB::Table",
            "State": "unknown",
            "StartTime": 1528317583.0,
            "EndTime": 1528317589.0,
            "Edges": [],
            "SummaryStatistics": {
                "OkCount": 2,
                "ErrorStatistics": {
                    "ThrottleCount": 0,
                    "OtherCount": 0,
                    "TotalCount": 0
                },
                "FaultStatistics": {
                    "OtherCount": 0,
                    "TotalCount": 0
                },
                "TotalCount": 2,
                "TotalResponseTime": 0.12
            },
            "DurationHistogram": [
                {
                    "Value": 0.076,
                    "Count": 1
                },
                {
                    "Value": 0.044,
                    "Count": 1
                }
            ],
            "ResponseTimeHistogram": [
                {
                    "Value": 0.076,
                    "Count": 1
                },
                {
                    "Value": 0.044,
                    "Count": 1
                }
            ]
        },
        {
            "ReferenceId": 2,
            "Name": "xray-sample.elasticbeanstalk.com",
            "Names": [
                "xray-sample.elasticbeanstalk.com"
            ],
            "Root": true,
            "Type": "AWS::EC2::Instance",
            "State": "active",
            "StartTime": 1528317567.0,
            "EndTime": 1528317589.0,
            "Edges": [
                {
                    "ReferenceId": 1,
                    "StartTime": 1528317567.0,
                    "EndTime": 1528317589.0,
                    "SummaryStatistics": {
                        "OkCount": 2,
                        "ErrorStatistics": {
                            "ThrottleCount": 0,
                            "OtherCount": 0,
                            "TotalCount": 0
                        },
                        "FaultStatistics": {
                            "OtherCount": 0,
                            "TotalCount": 0
                        },
                        "TotalCount": 2,
                        "TotalResponseTime": 0.12
                    },
                    "ResponseTimeHistogram": [
                        {
                            "Value": 0.076,
                            "Count": 1
                        },
                        {
                            "Value": 0.044,
                            "Count": 1
                        }
                    ],
                    "Aliases": []
                },
                {
                    "ReferenceId": 3,
                    "StartTime": 1528317567.0,
                    "EndTime": 1528317589.0,
                    "SummaryStatistics": {
                        "OkCount": 2,
                        "ErrorStatistics": {
                            "ThrottleCount": 0,
                            "OtherCount": 0,
                            "TotalCount": 0
                        },
                        "FaultStatistics": {
                            "OtherCount": 0,
                            "TotalCount": 0
                        },
                        "TotalCount": 2,
                        "TotalResponseTime": 0.125
                    },
                    "ResponseTimeHistogram": [
                        {
                            "Value": 0.049,
                            "Count": 1
                        },
                        {
                            "Value": 0.076,
                            "Count": 1
                        }
                    ],
                    "Aliases": []
                }
            ],
            "SummaryStatistics": {
                "OkCount": 3,
                "ErrorStatistics": {
                    "ThrottleCount": 0,
                    "OtherCount": 1,
                    "TotalCount": 1
                },
                "FaultStatistics": {
                    "OtherCount": 0,
                    "TotalCount": 0
                },
                "TotalCount": 4,
                "TotalResponseTime": 0.273
            },
            "DurationHistogram": [
                {
                    "Value": 0.005,
                    "Count": 1
                },
                {
                    "Value": 0.015,
                    "Count": 1
                },
                {
                    "Value": 0.157,
                    "Count": 1
                },
                {
                    "Value": 0.096,
                    "Count": 1
                }
            ],
            "ResponseTimeHistogram": [
                {
                    "Value": 0.005,
                    "Count": 1
                },
                {
                    "Value": 0.015,
                    "Count": 1
                },
                {
                    "Value": 0.157,
                    "Count": 1
                },
                {
                    "Value": 0.096,
                    "Count": 1
                }
            ]
        },
        {
            "ReferenceId": 3,
            "Name": "SNS",
            "Names": [
                "SNS"
            ],
            "Type": "AWS::SNS",
            "State": "unknown",
            "StartTime": 1528317583.0,
            "EndTime": 1528317589.0,
            "Edges": [],
            "SummaryStatistics": {
                "OkCount": 2,
                "ErrorStatistics": {
                    "ThrottleCount": 0,
                    "OtherCount": 0,
                    "TotalCount": 0
                },
                "FaultStatistics": {
                    "OtherCount": 0,
                    "TotalCount": 0
                },
                "TotalCount": 2,
                "TotalResponseTime": 0.125
            },
            "DurationHistogram": [
                {
                    "Value": 0.049,
                    "Count": 1
                },
                {
                    "Value": 0.076,
                    "Count": 1
                }
            ],
            "ResponseTimeHistogram": [
                {
                    "Value": 0.049,
                    "Count": 1
                },
                {
                    "Value": 0.076,
                    "Count": 1
                }
            ]
        }
    ]
```

```
}
```

Retrieving the service graph by group

To call for a service graph based on the contents of a group, include a `groupName` or `groupARN`. The following example shows a service graph call to a group named Example1.

Example Script to retrieve a service graph by name for group Example1

```
aws xray get-service-graph --group-name "Example1"
```

Retrieving traces

You can use the [`GetTraceSummaries`] API to get a list of trace summaries. Trace summaries include information that you can use to identify traces that you want to download in full, including annotations, request and response information, and IDs.

There are two `TimeRangeType` flags available when calling `aws xray get-trace-summaries`:

- **TraceId –** The default `GetTraceSummaries` search uses TraceID time and returns traces started within the computed `[start_time, end_time)` range. This range of timestamps is calculated based on the encoding of the timestamp within the TraceId, or can be defined manually.
- **Event time –** To search for events as they happen over the time, AWS X-Ray allows searching for traces using event timestamps. Event time returns traces active during the `[start_time, end_time)` range, regardless of when the trace began.

Use the `aws xray get-trace-summaries` command to get a list of trace summaries. The following commands get a list of trace summaries from between 1 and 2 minutes in the past using the default TraceId time.

Example Script to get trace summaries

```
EPOCH=$(date +%s)
aws xray get-trace-summaries --start-time $(($EPOCH-120)) --end-time $(($EPOCH-60))
```

Example GetTraceSummaries output

```
{
    "TraceSummaries": [
        {
            "HasError": false,
            "Http": {
                "HttpStatus": 200,
                "ClientIp": "205.255.255.183",
                "HttpURL": "http://scorekeep.elasticbeanstalk.com/api/session",
                "UserAgent": "Mozilla/5.0 (Windows NT 6.1; Win64; x64)
 AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36",
                "HttpMethod": "POST"
            },
            "Users": [],
            "HasFault": false,
            "Annotations": {},
            "ResponseTime": 0.084,
            "Duration": 0.084,
            "Id": "1-59602606-a43a1ac52fc7ee0eea12a82c",
            "HasThrottle": false
        },
        {
            "HasError": false,
            "Http": {
                "HttpStatus": 200,
                "ClientIp": "205.255.255.183",
                "HttpURL": "http://scorekeep.elasticbeanstalk.com/api/user",
                "UserAgent": "Mozilla/5.0 (Windows NT 6.1; Win64; x64)
 AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36",
                "HttpMethod": "POST"
            },
            "Users": [
                {
                    "UserName": "5M388M1E"
                }
            ],
            "HasFault": false,
            "Annotations": {
                "UserID": [
                    {
                        "AnnotationValue": {
                            "StringValue": "5M388M1E"
                        }
                    }
                ],
                "Name": [
                    {
                        "AnnotationValue": {
                            "StringValue": "Ola"
                        }
                    }
                ]
            },
            "ResponseTime": 3.232,
            "Duration": 3.232,
            "Id": "1-59602603-23fc5b688855d396af79b496",
            "HasThrottle": false
        }
    ],
    "ApproximateTime": 1499473304.0,
    "TracesProcessedCount": 2
}
```

Use the trace ID from the output to retrieve a full trace with the `BatchGetTraces` API.

Example BatchGetTraces command

```
$ aws xray batch-get-traces --trace-ids 1-596025b4-7170afe49f7aa708b1dd4a6b
```

Example BatchGetTraces output

```
{
    "Traces": [
        {
            "Duration": 3.232,
            "Segments": [
                {
                    "Document": "{\"id\":\"1fb07842d944e714\",\"name\":
\"random-name\",\"start_time\":1.499473411677E9,\"end_time\":1.499473414572E9,
\"parent_id\":\"0c544c1b1bbff948\",\"http\":{\"response\":{\"status\":200}},
\"aws\":{\"request_id\":\"ac086670-6373-11e7-a174-f31b3397f190\"},\"trace_id\":
\"1-59602603-23fc5b688855d396af79b496\",\"origin\":\"AWS::Lambda\",\"resource_arn\":
\"arn:aws:lambda:us-west-2:123456789012:function:random-name\"}",
                    "Id": "1fb07842d944e714"
                },
                {
                    "Document": "{\"id\":\"194fcc8747581230\",\"name\":\"Scorekeep
\",\"start_time\":1.499473411562E9,\"end_time\":1.499473414794E9,\"http\":{\"request
\":{\"url\":\"http://scorekeep.elasticbeanstalk.com/api/user\",\"method\":\"POST\",
\"user_agent\":\"Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML,
 like Gecko) Chrome/59.0.3071.115 Safari/537.36\",\"client_ip\":\"205.251.233.183\"},
\"response\":{\"status\":200}},\"aws\":{\"elastic_beanstalk\":{\"version_label\":\"app-
abb9-170708_002045\",\"deployment_id\":406,\"environment_name\":\"scorekeep-dev\"},
\"ec2\":{\"availability_zone\":\"us-west-2c\",\"instance_id\":\"i-0cd9e448944061b4a
\"},\"xray\":{\"sdk_version\":\"1.1.2\",\"sdk\":\"X-Ray for Java\"}},\"service
\":{},\"trace_id\":\"1-59602603-23fc5b688855d396af79b496\",\"user\":\"5M388M1E
\",\"origin\":\"AWS::ElasticBeanstalk::Environment\",\"subsegments\":[{\"id\":
\"0c544c1b1bbff948\",\"name\":\"Lambda\",\"start_time\":1.499473411629E9,\"end_time
\":1.499473414572E9,\"http\":{\"response\":{\"status\":200,\"content_length\":14}},
\"aws\":{\"log_type\":\"None\",\"status_code\":200,\"function_name\":\"random-name
\",\"invocation_type\":\"RequestResponse\",\"operation\":\"Invoke\",\"request_id
\":\"ac086670-6373-11e7-a174-f31b3397f190\",\"resource_names\":[\"random-name\"]},
\"namespace\":\"aws\"},{\"id\":\"071684f2e555e571\",\"name\":\"## UserModel.saveUser
\",\"start_time\":1.499473414581E9,\"end_time\":1.499473414769E9,\"metadata\":{\"debug
\":{\"test\":\"Metadata string from UserModel.saveUser\"}},\"subsegments\":[{\"id\":
\"4cd3f10b76c624b4\",\"name\":\"DynamoDB\",\"start_time\":1.49947341469E9,\"end_time
\":1.499473414769E9,\"http\":{\"response\":{\"status\":200,\"content_length\":57}},
\"aws\":{\"table_name\":\"scorekeep-user\",\"operation\":\"UpdateItem\",\"request_id
\":\"MFQ8CGJ3JTDDVVVASUAAJGQ6NJ82F738BOB4KQNSO5AEMVJF66Q9\",\"resource_names\":
[\"scorekeep-user\"]},\"namespace\":\"aws\"}]}]}",
                    "Id": "194fcc8747581230"
                },
                {
                    "Document": "{\"id\":\"00f91aa01f4984fd\",\"name\":
\"random-name\",\"start_time\":1.49947341283E9,\"end_time\":1.49947341457E9,
\"parent_id\":\"1fb07842d944e714\",\"aws\":{\"function_arn\":\"arn:aws:lambda:us-
west-2:123456789012:function:random-name\",\"resource_names\":[\"random-name\"],
\"account_id\":\"123456789012\"},\"trace_id\":\"1-59602603-23fc5b688855d396af79b496\",
\"origin\":\"AWS::Lambda::Function\",\"subsegments\":[{\"id\":\"e6d2fe619f827804\",
\"name\":\"annotations\",\"start_time\":1.499473413012E9,\"end_time\":1.499473413069E9,
\"annotations\":{\"UserID\":\"5M388M1E\",\"Name\":\"Ola\"}},{\"id\":\"b29b548af4d54a0f
\",\"name\":\"SNS\",\"start_time\":1.499473413112E9,\"end_time\":1.499473414071E9,
\"http\":{\"response\":{\"status\":200}},\"aws\":{\"operation\":\"Publish\",
\"region\":\"us-west-2\",\"request_id\":\"a2137970-f6fc-5029-83e8-28aadeb99198\",
\"retries\":0,\"topic_arn\":\"arn:aws:sns:us-west-2:123456789012:awseb-e-
ruag3jyweb-stack-NotificationTopic-6B829NT9V5O9\"},\"namespace\":\"aws\"},{\"id\":
\"2279c0030c955e52\",\"name\":\"Initialization\",\"start_time\":1.499473412064E9,
\"end_time\":1.499473412819E9,\"aws\":{\"function_arn\":\"arn:aws:lambda:us-
west-2:123456789012:function:random-name\"}}]}",
                    "Id": "00f91aa01f4984fd"
                },
                {
                    "Document": "{\"id\":\"17ba309b32c7fbaf\",\"name\":
\"DynamoDB\",\"start_time\":1.49947341469E9,\"end_time\":1.499473414769E9,
\"parent_id\":\"4cd3f10b76c624b4\",\"inferred\":true,\"http\":{\"response
```

```
\":{\"status\":200,\"content_length\":57}},\"aws\":{\"table_name
\":\"scorekeep-user\",\"operation\":\"UpdateItem\",\"request_id\":
\"MFQ8CGJ3JTDDVVVASUAAJGQ6NJ82F738BOB4KQNSO5AEMVJF66Q9\",\"resource_names\":
[\"scorekeep-user\"]},\"trace_id\":\"1-59602603-23fc5b688855d396af79b496\",\"origin\":
\"AWS::DynamoDB::Table\"}",
                    "Id": "17ba309b32c7fbaf"
                },
                {
                    "Document": "{\"id\":\"1ee3c4a523f89ca5\",\"name\":\"SNS
\",\"start_time\":1.499473413112E9,\"end_time\":1.499473414071E9,\"parent_id\":
\"b29b548af4d54a0f\",\"inferred\":true,\"http\":{\"response\":{\"status\":200}},\"aws
\":{\"operation\":\"Publish\",\"region\":\"us-west-2\",\"request_id\":\"a2137970-
f6fc-5029-83e8-28aadeb99198\",\"retries\":0,\"topic_arn\":\"arn:aws:sns:us-
west-2:123456789012:awseb-e-ruag3jyweb-stack-NotificationTopic-6B829NT9V5O9\"},
\"trace_id\":\"1-59602603-23fc5b688855d396af79b496\",\"origin\":\"AWS::SNS\"}",
                    "Id": "1ee3c4a523f89ca5"
                }
            ],
            "Id": "1-59602603-23fc5b688855d396af79b496"
        }
    ],
    "UnprocessedTraceIds": []
}
```

The full trace includes a document for each segment, compiled from all of the segment documents received with the same trace ID. These documents don't represent the data as it was sent to X-Ray by your application. Instead, they represent the processed documents generated by the X-Ray service. X-Ray creates the full trace document by compiling segment documents sent by your application, and removing data that doesn't comply with the segment document schema.

X-Ray also creates *inferred segments* for downstream calls to services that don't send segments themselves. For example, when you call DynamoDB with an instrumented client, the X-Ray SDK records a subsegment with details about the call from its point of view. However, DynamoDB doesn't send a corresponding segment. X-Ray uses the information in the subsegment to create an inferred segment to represent the DynamoDB resource in the trace map, and adds it to the trace document.

To get multiple traces from the API, you need a list of trace IDs, which you can extract from the output of `get-trace-summaries` with an AWS CLI query. Redirect the list to the input of `batch-get-traces` to get full traces for a specific time period.

Example Script to get full traces for a one minute period

```
EPOCH=$(date +%s)
TRACEIDS=$(aws xray get-trace-summaries --start-time $(($EPOCH-120)) --end-time
 $(($EPOCH-60)) --query 'TraceSummaries[*].Id' --output text)
aws xray batch-get-traces --trace-ids $TRACEIDS --query 'Traces[*]'
```

Retrieving and refining root cause analytics

Upon generating a trace summary with the GetTraceSummaries API , partial trace summaries can be reused in their JSON format to create a refined filter expression based upon root causes. See the examples below for a walkthrough of the refinement steps.

Example Example GetTraceSummaries output - response time root cause section

```
{
  "Services": [
    {
      "Name": "GetWeatherData",
      "Names": ["GetWeatherData"],
      "AccountId": 123456789012,
      "Type": null,
      "Inferred": false,
      "EntityPath": [
        {
          "Name": "GetWeatherData",
          "Coverage": 1.0,
          'Remote": false
        },
        {
          "Name": "get_temperature",
          "Coverage": 0.8,
          "Remote": false
        }
      ]
    },
    {
      "Name": "GetTemperature",
      "Names": ["GetTemperature"],
      "AccountId": 123456789012,
      "Type": null,
      "Inferred": false,
      "EntityPath": [
        {
          "Name": "GetTemperature",
          "Coverage": 0.7,
          "Remote": false
        }
      ]
    }
  ]
}
```

By editing and making omissions to the above output, this JSON can become a filter for matched root cause entities. For every field present in the JSON, any candidate match must be exact, or the trace will not be returned. Removed fields become wildcard values, a format which is compatible with the filter expression query structure.

Example Reformatted response time root cause

```
{
  "Services": [
    {
      "Name": "GetWeatherData",
      "EntityPath": [
        {
          "Name": "GetWeatherData"
        },
        {
          "Name": "get_temperature"
        }
      ]
    },
    {
      "Name": "GetTemperature",
      "EntityPath": [
        {
          "Name": "GetTemperature"
        }
      ]
    }
  ]
}
```

This JSON is then used as part of a filter expression through a call to `rootcause.json = #[{}]`. Refer to the Filter Expressions chapter for more details about querying with filter expressions.

Example Example JSON filter

```
rootcause.json = #[{ "Services": [ { "Name": "GetWeatherData", "EntityPath": [{ "Name":
 "GetWeatherData" }, { "Name": "get_temperature" } ] }, { "Name": "GetTemperature",
 "EntityPath": [ { "Name": "GetTemperature" } ] } ] }]
```

Configuring sampling, groups, and encryption settings with the AWS X-Ray API

AWS X-Ray provides APIs for configuring sampling rules, group rules, and encryption settings.

Sections

- Encryption settings
- Sampling rules
- Groups

Encryption settings

Use `PutEncryptionConfig` to specify an AWS Key Management Service (AWS KMS) key to use for encryption.

> **Note**
>
> X-Ray does not support asymmetric KMS keys.

```
$ aws xray put-encryption-config --type KMS --key-id alias/aws/xray
{
    "EncryptionConfig": {
        "KeyId": "arn:aws:kms:us-east-2:123456789012:key/c234g4e8-39e9-4gb0-84e2-
b0ea215cbba5",
        "Status": "UPDATING",
        "Type": "KMS"
    }
}
```

For the key ID, you can use an alias (as shown in the example), a key ID, or an Amazon Resource Name (ARN).

Use `GetEncryptionConfig` to get the current configuration. When X-Ray finishes applying your settings, the status changes from `UPDATING` to `ACTIVE`.

```
$ aws xray get-encryption-config
{
    "EncryptionConfig": {
        "KeyId": "arn:aws:kms:us-east-2:123456789012:key/c234g4e8-39e9-4gb0-84e2-
b0ea215cbba5",
        "Status": "ACTIVE",
        "Type": "KMS"
    }
}
```

To stop using a KMS key and use default encryption, set the encryption type to `NONE`.

```
$ aws xray put-encryption-config --type NONE
{
    "EncryptionConfig": {
        "Status": "UPDATING",
        "Type": "NONE"
    }
}
```

Sampling rules

You can manage the sampling rules in your account with the X-Ray API. For more information about adding and managing tags, see Tagging X-Ray sampling rules and groups.

Get all sampling rules with `GetSamplingRules`.

```
$ aws xray get-sampling-rules
{
    "SamplingRuleRecords": [
        {
            "SamplingRule": {
                "RuleName": "Default",
                "RuleARN": "arn:aws:xray:us-east-2:123456789012:sampling-rule/Default",
                "ResourceARN": "*",
                "Priority": 10000,
                "FixedRate": 0.05,
                "ReservoirSize": 1,
                "ServiceName": "*",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "*",
                "URLPath": "*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 0.0,
            "ModifiedAt": 1529959993.0
        }
    ]
}
```

The default rule applies to all requests that don't match another rule. It is the lowest priority rule and cannot be deleted. You can, however, change the rate and reservoir size with UpdateSamplingRule.

Example API input for UpdateSamplingRule – 10000-default.json

```
{
    "SamplingRuleUpdate": {
        "RuleName": "Default",
        "FixedRate": 0.01,
        "ReservoirSize": 0
    }
}
```

The following example uses the previous file as input to change the default rule to one percent with no reservoir. Tags are optional. If you choose to add tags, a tag key is required, and tag values are optional. To remove existing tags from a sampling rule, use UntagResource

```
$ aws xray update-sampling-rule --cli-input-json file://1000-default.json --tags
 [{"Key": "key_name","Value": "value"},{"Key": "key_name","Value": "value"}]
{
    "SamplingRuleRecords": [
        {
            "SamplingRule": {
                "RuleName": "Default",
                "RuleARN": "arn:aws:xray:us-east-2:123456789012:sampling-rule/Default",
                "ResourceARN": "*",
                "Priority": 10000,
                "FixedRate": 0.01,
                "ReservoirSize": 0,
                "ServiceName": "*",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "*",
                "URLPath": "*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 0.0,
            "ModifiedAt": 1529959993.0
        },
```

Create additional sampling rules with `CreateSamplingRule`. When you create a rule, most of the rule fields are required. The following example creates two rules. This first rule sets a base rate for the Scorekeep sample application. It matches all requests served by the API that don't match a higher priority rule.

Example API input for `UpdateSamplingRule` – 9000-base-scorekeep.json

```
{
    "SamplingRule": {
        "RuleName": "base-scorekeep",
        "ResourceARN": "*",
        "Priority": 9000,
        "FixedRate": 0.1,
        "ReservoirSize": 5,
        "ServiceName": "Scorekeep",
        "ServiceType": "*",
        "Host": "*",
        "HTTPMethod": "*",
        "URLPath": "*",
        "Version": 1
    }
}
```

The second rule also applies to Scorekeep, but it has a higher priority and is more specific. This rule sets a very low sampling rate for polling requests. These are GET requests made by the client every few seconds to check for changes to the game state.

Example API input for UpdateSamplingRule – 5000-polling-scorekeep.json

```
{
    "SamplingRule": {
        "RuleName": "polling-scorekeep",
        "ResourceARN": "*",
        "Priority": 5000,
        "FixedRate": 0.003,
        "ReservoirSize": 0,
        "ServiceName": "Scorekeep",
        "ServiceType": "*",
        "Host": "*",
        "HTTPMethod": "GET",
        "URLPath": "/api/state/*",
        "Version": 1
    }
}
```

Tags are optional. If you choose to add tags, a tag key is required, and tag values are optional.

```
$ aws xray create-sampling-rule --cli-input-json file://5000-polling-scorekeep.json --
tags [{"Key": "key_name","Value": "value"},{"Key": "key_name","Value": "value"}]
{
    "SamplingRuleRecord": {
        "SamplingRule": {
            "RuleName": "polling-scorekeep",
            "RuleARN": "arn:aws:xray:us-east-1:123456789012:sampling-rule/polling-
scorekeep",
            "ResourceARN": "*",
            "Priority": 5000,
            "FixedRate": 0.003,
            "ReservoirSize": 0,
            "ServiceName": "Scorekeep",
            "ServiceType": "*",
            "Host": "*",
            "HTTPMethod": "GET",
            "URLPath": "/api/state/*",
            "Version": 1,
            "Attributes": {}
        },
        "CreatedAt": 1530574399.0,
        "ModifiedAt": 1530574399.0
    }
}
$ aws xray create-sampling-rule --cli-input-json file://9000-base-scorekeep.json
{
    "SamplingRuleRecord": {
        "SamplingRule": {
            "RuleName": "base-scorekeep",
            "RuleARN": "arn:aws:xray:us-east-1:123456789012:sampling-rule/base-
scorekeep",
            "ResourceARN": "*",
            "Priority": 9000,
            "FixedRate": 0.1,
            "ReservoirSize": 5,
            "ServiceName": "Scorekeep",
            "ServiceType": "*",
            "Host": "*",
            "HTTPMethod": "*",
            "URLPath": "*",
            "Version": 1,
            "Attributes": {}
        },
        "CreatedAt": 1530574410.0,
        "ModifiedAt": 1530574410.0
    }
}
```

To delete a sampling rule, use `DeleteSamplingRule`.

```
$ aws xray delete-sampling-rule --rule-name polling-scorekeep
{
    "SamplingRuleRecord": {
        "SamplingRule": {
            "RuleName": "polling-scorekeep",
            "RuleARN": "arn:aws:xray:us-east-1:123456789012:sampling-rule/polling-
scorekeep",
            "ResourceARN": "*",
            "Priority": 5000,
            "FixedRate": 0.003,
            "ReservoirSize": 0,
            "ServiceName": "Scorekeep",
            "ServiceType": "*",
            "Host": "*",
            "HTTPMethod": "GET",
            "URLPath": "/api/state/*",
            "Version": 1,
            "Attributes": {}
        },
        "CreatedAt": 1530574399.0,
        "ModifiedAt": 1530574399.0
    }
}
```

Groups

You can use the X-Ray API to manage groups in your account. Groups are a collection of traces that are defined by a filter expression. You can use groups to generate additional service graphs and supply Amazon CloudWatch metrics. See Getting data from AWS X-Ray for more details about working with service graphs and metrics through the X-Ray API. For more information about groups, see Configuring groups. For more information about adding and managing tags, see Tagging X-Ray sampling rules and groups.

Create a group with `CreateGroup`. Tags are optional. If you choose to add tags, a tag key is required, and tag values are optional.

```
$ aws xray create-group --group-name "TestGroup" --filter-expression
 "service(\"example.com\") {fault}" --tags [{"Key": "key_name","Value": "value"},
{"Key": "key_name","Value": "value"}]
{
    "GroupName": "TestGroup",
    "GroupARN": "arn:aws:xray:us-east-2:123456789012:group/TestGroup/UniqueID",
    "FilterExpression": "service(\"example.com\") {fault OR error}"
}
```

Get all existing groups with `GetGroups`.

```
$ aws xray get-groups
{
    "Groups": [
        {
            "GroupName": "TestGroup",
            "GroupARN": "arn:aws:xray:us-east-2:123456789012:group/TestGroup/UniqueID",
            "FilterExpression": "service(\"example.com\") {fault OR error}"
        },
  {
            "GroupName": "TestGroup2",
            "GroupARN": "arn:aws:xray:us-east-2:123456789012:group/TestGroup2/
UniqueID",
            "FilterExpression": "responsetime > 2"
        }
    ],
 "NextToken": "tokenstring"
}
```

Update a group with `UpdateGroup`. Tags are optional. If you choose to add tags, a tag key is required, and tag values are optional. To remove existing tags from a group, use UntagResource.

```
$ aws xray update-group --group-name "TestGroup" --group-arn "arn:aws:xray:us-
east-2:123456789012:group/TestGroup/UniqueID" --filter-expression
 "service(\"example.com\") {fault OR error}" --tags [{"Key": "Stage","Value": "Prod"},
{"Key": "Department","Value": "QA"}]
{
    "GroupName": "TestGroup",
    "GroupARN": "arn:aws:xray:us-east-2:123456789012:group/TestGroup/UniqueID",
    "FilterExpression": "service(\"example.com\") {fault OR error}"
}
```

Delete a group with `DeleteGroup`.

```
$ aws xray delete-group --group-name "TestGroup" --group-arn "arn:aws:xray:us-
east-2:123456789012:group/TestGroup/UniqueID"
    {
    }
```

Using sampling rules with the X-Ray API

The AWS X-Ray SDK uses the X-Ray API to get sampling rules, report sampling results, and get quotas. You can use these APIs to get a better understanding of how sampling rules work, or to implement sampling in a language that the X-Ray SDK doesn't support.

Start by getting all sampling rules with `GetSamplingRules`.

```
$ aws xray get-sampling-rules
{
    "SamplingRuleRecords": [
        {
            "SamplingRule": {
                "RuleName": "Default",
                "RuleARN": "arn:aws:xray:us-east-1::sampling-rule/Default",
                "ResourceARN": "*",
                "Priority": 10000,
                "FixedRate": 0.01,
                "ReservoirSize": 0,
                "ServiceName": "*",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "*",
                "URLPath": "*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 0.0,
            "ModifiedAt": 1530558121.0
        },
        {
            "SamplingRule": {
                "RuleName": "base-scorekeep",
                "RuleARN": "arn:aws:xray:us-east-1::sampling-rule/base-scorekeep",
                "ResourceARN": "*",
                "Priority": 9000,
                "FixedRate": 0.1,
                "ReservoirSize": 2,
                "ServiceName": "Scorekeep",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "*",
                "URLPath": "*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 1530573954.0,
            "ModifiedAt": 1530920505.0
        },
        {
            "SamplingRule": {
                "RuleName": "polling-scorekeep",
                "RuleARN": "arn:aws:xray:us-east-1::sampling-rule/polling-scorekeep",
                "ResourceARN": "*",
                "Priority": 5000,
                "FixedRate": 0.003,
                "ReservoirSize": 0,
                "ServiceName": "Scorekeep",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "GET",
                "URLPath": "/api/state/*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 1530918163.0,
            "ModifiedAt": 1530918163.0
        }
    ]
}
```

The output includes the default rule and custom rules. See Sampling rules if you haven't yet created sampling rules.

Evaluate rules against incoming requests in ascending order of priority. When a rule matches, use the fixed rate and reservoir size to make a sampling decision. Record sampled requests and ignore (for tracing purposes) unsampled requests. Stop evaluating rules when a sampling decision is made.

A rules reservoir size is the target number of traces to record per second before applying the fixed rate. The reservoir applies across all services cumulatively, so you can't use it directly. However, if it is non-zero, you can borrow one trace per second from the reservoir until X-Ray assigns a quota. Before receiving a quota, record the first request each second, and apply the fixed rate to additional requests. The fixed rate is a decimal between 0 and 1.00 (100%).

The following example shows a call to `GetSamplingTargets` with details about sampling decisions made over the last 10 seconds.

```
$ aws xray get-sampling-targets --sampling-statistics-documents '[
    {
        "RuleName": "base-scorekeep",
        "ClientID": "ABCDEF1234567890ABCDEF10",
        "Timestamp": "2018-07-07T00:20:06",
        "RequestCount": 110,
        "SampledCount": 20,
        "BorrowCount": 10
    },
    {
        "RuleName": "polling-scorekeep",
        "ClientID": "ABCDEF1234567890ABCDEF10",
        "Timestamp": "2018-07-07T00:20:06",
        "RequestCount": 10500,
        "SampledCount": 31,
        "BorrowCount": 0
    }
]'
{
    "SamplingTargetDocuments": [
        {
            "RuleName": "base-scorekeep",
            "FixedRate": 0.1,
            "ReservoirQuota": 2,
            "ReservoirQuotaTTL": 1530923107.0,
            "Interval": 10
        },
        {
            "RuleName": "polling-scorekeep",
            "FixedRate": 0.003,
            "ReservoirQuota": 0,
            "ReservoirQuotaTTL": 1530923107.0,
            "Interval": 10
        }
    ],
    "LastRuleModification": 1530920505.0,
    "UnprocessedStatistics": []
}
```

The response from X-Ray includes a quota to use instead of borrowing from the reservoir. In this example, the service borrowed 10 traces from the reservoir over 10 seconds, and applied the fixed rate of 10 percent to the other 100 requests, resulting in a total of 20 sampled requests. The quota is good for five minutes (indicated by the time to live) or until a new quota is assigned. X-Ray may also assign a longer reporting interval than the default, although it didn't here.

Note

The response from X-Ray might not include a quota the first time you call it. Continue borrowing from the reservoir until you are assigned a quota.

The other two fields in the response might indicate issues with the input. Check LastRuleModification against the last time you called GetSamplingRules. If it's newer, get a new copy of the rules. UnprocessedStatistics can include errors that indicate that a rule has been deleted, that the statistics document in the input was too old, or permissions errors.

AWS X-Ray segment documents

A **trace segment** is a JSON representation of a request that your application serves. A trace segment records information about the original request, information about the work that your application does locally, and **subsegments** with information about downstream calls that your application makes to AWS resources, HTTP APIs, and SQL databases.

A **segment document** conveys information about a segment to X-Ray. A segment document can be up to 64 kB and contain a whole segment with subsegments, a fragment of a segment that indicates that a request is in progress, or a single subsegment that is sent separately. You can send segment documents directly to X-Ray by using the PutTraceSegments API.

X-Ray compiles and processes segment documents to generate queryable **trace summaries** and **full traces** that you can access by using the GetTraceSummaries and BatchGetTraces APIs, respectively. In addition to the segments and subsegments that you send to X-Ray, the service uses information in subsegments to generate **inferred segments** and adds them to the full trace. Inferred segments represent downstream services and resources in the trace map.

X-Ray provides a **JSON schema** for segment documents. You can download the schema here: xray-segmentdocument-schema-v1.0.0. The fields and objects listed in the schema are described in more detail in the following sections.

A subset of segment fields are indexed by X-Ray for use with filter expressions. For example, if you set the user field on a segment to a unique identifier, you can search for segments associated with specific users in the X-Ray console or by using the GetTraceSummaries API. For more information, see Using filter expressions.

When you instrument your application with the X-Ray SDK, the SDK generates segment documents for you. Instead of sending segment documents directly to X-Ray, the SDK transmits them over a local UDP port to the X-Ray daemon. For more information, see Sending segment documents to the X-Ray daemon.

Sections

- Segment fields

- Subsegments
- HTTP request data
- Annotations
- Metadata
- AWS resource data
- Errors and exceptions
- SQL queries

Segment fields

A segment records tracing information about a request that your application serves. At a minimum, a segment records the name, ID, start time, trace ID, and end time of the request.

Example Minimal complete segment

```
{
  "name" : "example.com",
  "id" : "70de5b6f19ff9a0a",
  "start_time" : 1.478293361271E9,
  "trace_id" : "1-581cf771-a006649127e371903a2de979",
  "end_time" : 1.478293361449E9
}
```

The following fields are required, or conditionally required, for segments.

> **Note**
>
> Values must be strings (up to 250 characters) unless noted otherwise.

Required Segment Fields

- `name` – The logical name of the service that handled the request, up to **200 characters**. For example, your application's name or domain name. Names can contain Unicode letters, numbers, and whitespace, and the following symbols: `_`, `.`, `:`, `/`, `%`, `&`, `#`, `=`, `+`, `\`, `-`, `@`
- `id` – A 64-bit identifier for the segment, unique among segments in the same trace, in **16 hexadecimal digits**.

- `trace_id` – A unique identifier that connects all segments and subsegments originating from a single client request.

  X-Ray trace ID format

  An X-Ray `trace_id` consists of three numbers separated by hyphens. For example, `1-58406520-a006649127e371903a2de979`. This includes:

  - The version number, which is `1`.
  - The time of the original request in Unix epoch time using **8 hexadecimal digits**.

    For example, 10:00AM December 1st, 2016 PST in epoch time is `1480615200` seconds or `58406520` in hexadecimal digits.
  - A globally unique 96-bit identifier for the trace in **24 hexadecimal digits**.

  > **Note**
  >
  > X-Ray now supports trace IDs that are created using OpenTelemetry and any other framework that conforms with the W3C Trace Context specification. A W3C trace ID must be formatted in X-Ray trace ID format when sending to X-Ray. For example, W3C trace ID `4efaaf4d1e8720b39541901950019ee5` should be formatted as `1-4efaaf4d-1e8720b39541901950019ee5` when sending to X-Ray. X-Ray trace IDs include the original request time stamp in Unix epoch time, but this isn't required when sending W3C trace IDs in X-Ray format.

  > **Trace ID Security**
  >
  > Trace IDs are visible in response headers. Generate trace IDs with a secure random algorithm to ensure that attackers cannot calculate future trace IDs and send requests with those IDs to your application.

- `start_time` – **number** that is the time the segment was created, in floating point seconds in epoch time. For example, `1480615200.010` or `1.480615200010E9`. Use as many decimal places as you need. Microsecond resolution is recommended when available.
- `end_time` – **number** that is the time the segment was closed. For example, `1480615200.090` or `1.480615200090E9`. Specify either an `end_time` or `in_progress`.

- `in_progress` – **boolean**, set to `true` instead of specifying an `end_time` to record that a segment is started, but is not complete. Send an in-progress segment when your application receives a request that will take a long time to serve, to trace the request receipt. When the response is sent, send the complete segment to overwrite the in-progress segment. Only send one complete segment, and one or zero in-progress segments, per request.

> **Service Names**
>
> A segment's `name` should match the domain name or logical name of the service that generates the segment. However, this is not enforced. Any application that has permission to `PutTraceSegments` can send segments with any name.

The following fields are optional for segments.

Optional Segment Fields

- `service` – An object with information about your application.
  - `version` – A string that identifies the version of your application that served the request.
- `user` – A string that identifies the user who sent the request.
- `origin` – The type of AWS resource running your application.

  Supported Values

  - `AWS::EC2::Instance` – An Amazon EC2 instance.
  - `AWS::ECS::Container` – An Amazon ECS container.
  - `AWS::ElasticBeanstalk::Environment` – An Elastic Beanstalk environment.

  When multiple values are applicable to your application, use the one that is most specific. For example, a Multicontainer Docker Elastic Beanstalk environment runs your application on an Amazon ECS container, which in turn runs on an Amazon EC2 instance. In this case you would set the origin to `AWS::ElasticBeanstalk::Environment` as the environment is the parent of the other two resources.
- `parent_id` – A subsegment ID you specify if the request originated from an instrumented application. The X-Ray SDK adds the parent subsegment ID to the tracing header for downstream HTTP calls. In the case of nested subsegments, a subsegment can have a segment or a subsegment as its parent.

- `http` – `http` objects with information about the original HTTP request.
- `aws` – `aws` object with information about the AWS resource on which your application served the request.
- `error`, `throttle`, `fault`, and `cause` – error fields that indicate an error occurred and that include information about the exception that caused the error.
- `annotations` – `annotations` object with key-value pairs that you want X-Ray to index for search.
- `metadata` – `metadata` object with any additional data that you want to store in the segment.
- `subsegments` – **array** of `subsegment` objects.

Subsegments

You can create subsegments to record calls to AWS services and resources that you make with the AWS SDK, calls to internal or external HTTP web APIs, or SQL database queries. You can also create subsegments to debug or annotate blocks of code in your application. Subsegments can contain other subsegments, so a custom subsegment that records metadata about an internal function call can contain other custom subsegments and subsegments for downstream calls.

A subsegment records a downstream call from the point of view of the service that calls it. X-Ray uses subsegments to identify downstream services that don't send segments and create entries for them on the service graph.

A subsegment can be embedded in a full segment document or sent independently. Send subsegments separately to asynchronously trace downstream calls for long-running requests, or to avoid exceeding the maximum segment document size.

Example Segment with embedded subsegment

An independent subsegment has a `type` of `subsegment` and a `parent_id` that identifies the parent segment.

```
{
  "trace_id"   : "1-5759e988-bd862e3fe1be46a994272793",
  "id"         : "defdfd9912dc5a56",
  "start_time" : 1461096053.37518,
  "end_time"   : 1461096053.4042,
  "name"       : "www.example.com",
  "http"       : {
    "request"  : {
      "url"        : "https://www.example.com/health",
      "method"     : "GET",
      "user_agent" : "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6)
 AppleWebKit/601.7.7",
      "client_ip"  : "11.0.3.111"
    },
    "response" : {
      "status"         : 200,
      "content_length" : 86
    }
  },
  "subsegments" : [
    {
      "id"         : "53995c3f42cd8ad8",
      "name"       : "api.example.com",
      "start_time" : 1461096053.37769,
      "end_time"   : 1461096053.40379,
      "namespace"  : "remote",
      "http"       : {
        "request"  : {
          "url"    : "https://api.example.com/health",
          "method" : "POST",
          "traced" : true
        },
        "response" : {
          "status"         : 200,
          "content_length" : 861
        }
      }
    }
  ]
}
```

For long-running requests, you can send an in-progress segment to notify X-Ray that the request was received, and then send subsegments separately to trace them before completing the original request.

Example In-progress segment

```
{
  "name" : "example.com",
  "id" : "70de5b6f19ff9a0b",
  "start_time" : 1.478293361271E9,
  "trace_id" : "1-581cf771-a006649127e371903a2de979",
  "in_progress": true
}
```

Example Independent subsegment

An independent subsegment has a `type` of `subsegment`, a `trace_id`, and a `parent_id` that identifies the parent segment.

```
{
  "name" : "api.example.com",
  "id" : "53995c3f42cd8ad8",
  "start_time" : 1.478293361271E9,
  "end_time" : 1.478293361449E9,
  "type" : "subsegment",
  "trace_id" : "1-581cf771-a006649127e371903a2de979"
  "parent_id" : "defdfd9912dc5a56",
  "namespace"  : "remote",
  "http"       : {
      "request"  : {
          "url"    : "https://api.example.com/health",
          "method" : "POST",
          "traced" : true
      },
      "response" : {
          "status"         : 200,
          "content_length" : 861
      }
  }
}
```

When the request is complete, close the segment by resending it with an `end_time`. The complete segment overwrites the in-progress segment.

You can also send subsegments separately for completed requests that triggered asynchronous workflows. For example, a web API may return a `OK 200` response immediately prior to starting the work that the user requested. You can send a full segment to X-Ray as soon as the response is sent, followed by subsegments for work completed later. As with segments, you can also send a subsegment fragment to record that the subsegment has started, and then overwrite it with a full subsegment once the downstream call is complete.

The following fields are required, or are conditionally required, for subsegments.

Note

Values are strings up to 250 characters unless noted otherwise.

Required Subsegment Fields

- `id` – A 64-bit identifier for the subsegment, unique among segments in the same trace, in **16 hexadecimal digits**.
- `name` – The logical name of the subsegment. For downstream calls, name the subsegment after the resource or service called. For custom subsegments, name the subsegment after the code that it instruments (e.g., a function name).
- `start_time` – **number** that is the time the subsegment was created, in floating point seconds in epoch time, accurate to milliseconds. For example, `1480615200.010` or `1.480615200010E9`.
- `end_time` – **number** that is the time the subsegment was closed. For example, `1480615200.090` or `1.480615200090E9`. Specify an `end_time` or `in_progress`.
- `in_progress` – **boolean** that is set to `true` instead of specifying an `end_time` to record that a subsegment is started, but is not complete. Only send one complete subsegment, and one or zero in-progress subsegments, per downstream request.
- `trace_id` – Trace ID of the subsegment's parent segment. Required only if sending a subsegment separately.

  #### X-Ray trace ID format

  An X-Ray `trace_id` consists of three numbers separated by hyphens. For example, `1-58406520-a006649127e371903a2de979`. This includes:

  - The version number, which is `1`.
  - The time of the original request in Unix epoch time using **8 hexadecimal digits**.

    For example, 10:00AM December 1st, 2016 PST in epoch time is `1480615200` seconds or `58406520` in hexadecimal digits.
  - A globally unique 96-bit identifier for the trace in **24 hexadecimal digits**.

  Note

  X-Ray now supports trace IDs that are created using OpenTelemetry and any other framework that conforms with the W3C Trace Context specification. A W3C trace ID

> must be formatted in X-Ray trace ID format when sending to X-Ray. For example, W3C trace ID `4efaaf4d1e8720b39541901950019ee5` should be formatted as `1-4efaaf4d-1e8720b39541901950019ee5` when sending to X-Ray. X-Ray trace IDs include the original request time stamp in Unix epoch time, but this isn't required when sending W3C trace IDs in X-Ray format.

- `parent_id` – Segment ID of the subsegment's parent segment. Required only if sending a subsegment separately. In the case of nested subsegments, a subsegment can have a segment or a subsegment as its parent.
- `type` – `subsegment`. Required only if sending a subsegment separately.

The following fields are optional for subsegments.

Optional Subsegment Fields

- `namespace` – `aws` for AWS SDK calls; `remote` for other downstream calls.
- `http` – `http` object with information about an outgoing HTTP call.
- `aws` – `aws` object with information about the downstream AWS resource that your application called.
- `error`, `throttle`, `fault`, and `cause` – error fields that indicate an error occurred and that include information about the exception that caused the error.
- `annotations` – `annotations` object with key-value pairs that you want X-Ray to index for search.
- `metadata` – `metadata` object with any additional data that you want to store in the segment.
- `subsegments` – **array** of `subsegment` objects.
- `precursor_ids` – **array** of subsegment IDs that identifies subsegments with the same parent that completed prior to this subsegment.

HTTP request data

Use an HTTP block to record details about an HTTP request that your application served (in a segment) or that your application made to a downstream HTTP API (in a subsegment). Most of the fields in this object map to information found in an HTTP request and response.

`http`

All fields are optional.

- `request` – Information about a request.
  - `method` – The request method. For example, `GET`.
  - `url` – The full URL of the request, compiled from the protocol, hostname, and path of the request.
  - `user_agent` – The user agent string from the requester's client.
  - `client_ip` – The IP address of the requester. Can be retrieved from the IP packet's `Source Address` or, for forwarded requests, from an `X-Forwarded-For` header.
  - `x_forwarded_for` – (segments only) **boolean** indicating that the `client_ip` was read from an `X-Forwarded-For` header and is not reliable as it could have been forged.
  - `traced` – (subsegments only) **boolean** indicating that the downstream call is to another traced service. If this field is set to `true`, X-Ray considers the trace to be broken until the downstream service uploads a segment with a `parent_id` that matches the `id` of the subsegment that contains this block.
- `response` – Information about a response.
  - `status` – **integer** indicating the HTTP status of the response.
  - `content_length` – **integer** indicating the length of the response body in bytes.

When you instrument a call to a downstream web api, record a subsegment with information about the HTTP request and response. X-Ray uses the subsegment to generate an inferred segment for the remote API.

Example Segment for HTTP call served by an application running on Amazon EC2

```
{
  "id": "6b55dcc497934f1a",
  "start_time": 1484789387.126,
  "end_time": 1484789387.535,
  "trace_id": "1-5880168b-fd5158284b67678a3bb5a78c",
  "name": "www.example.com",
  "origin": "AWS::EC2::Instance",
  "aws": {
    "ec2": {
      "availability_zone": "us-west-2c",
      "instance_id": "i-0b5a4678fc325bg98"
    },
    "xray": {
        "sdk_version": "2.11.0 for Java"
    },
  },
  "http": {
    "request": {
      "method": "POST",
      "client_ip": "78.255.233.48",
      "url": "http://www.example.com/api/user",
      "user_agent": "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:45.0) Gecko/20100101
 Firefox/45.0",
      "x_forwarded_for": true
    },
    "response": {
      "status": 200
    }
  }
```

Example Subsegment for a downstream HTTP call

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

Example Inferred segment for a downstream HTTP call

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
  "trace_id": "1-62be1272-1b71c4274f39f122afa64eab",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  },
  "inferred": true
}
```

Annotations

Segments and subsegments can include an `annotations` object containing one or more fields that X-Ray indexes for use with filter expressions. Fields can have string, number, or Boolean values (no objects or arrays). X-Ray indexes up to 50 annotations per trace.

Example Segment for HTTP call with annotations

```
{
  "id": "6b55dcc497932f1a",
  "start_time": 1484789187.126,
  "end_time": 1484789187.535,
  "trace_id": "1-5880168b-fd515828bs07678a3bb5a78c",
  "name": "www.example.com",
  "origin": "AWS::EC2::Instance",
  "aws": {
    "ec2": {
      "availability_zone": "us-west-2c",
      "instance_id": "i-0b5a4678fc325bg98"
    },
    "xray": {
        "sdk_version": "2.11.0 for Java"
    },
  },
  "annotations": {
    "customer_category" : 124,
    "zip_code" : 98101,
    "country" : "United States",
    "internal" : false
  },
  "http": {
    "request": {
      "method": "POST",
      "client_ip": "78.255.233.48",
      "url": "http://www.example.com/api/user",
      "user_agent": "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:45.0) Gecko/20100101
 Firefox/45.0",
      "x_forwarded_for": true
    },
    "response": {
      "status": 200
    }
  }
```

Keys must be alphanumeric in order to work with filters. Underscore is allowed. Other symbols and whitespace are not allowed.

Metadata

Segments and subsegments can include a `metadata` object containing one or more fields with values of any type, including objects and arrays. X-Ray does not index metadata, and values can be any size, as long as the segment document doesn't exceed the maximum size (64 kB). You can view metadata in the full segment document returned by the `BatchGetTraces` API. Field keys (`debug` in the following example) starting with `AWS.` are reserved for use by AWS-provided SDKs and clients.

Example Custom subsegment with metadata

```
{
  "id": "0e58d2918e9038e8",
  "start_time": 1484789387.502,
  "end_time": 1484789387.534,
  "name": "## UserModel.saveUser",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
  "subsegments": [
    {
      "id": "0f910026178b71eb",
      "start_time": 1484789387.502,
      "end_time": 1484789387.534,
      "name": "DynamoDB",
      "namespace": "aws",
      "http": {
        "response": {
          "content_length": 58,
          "status": 200
        }
      },
      "aws": {
        "table_name": "scorekeep-user",
        "operation": "UpdateItem",
        "request_id": "3AIENM5J4ELQ3SPODHKBIRVIC3VV4KQNSO5AEMVJF66Q9ASUAAJG",
        "resource_names": [
          "scorekeep-user"
        ]
      }
    }
  ]
}
```

AWS resource data

For segments, the `aws` object contains information about the resource on which your application is running. Multiple fields can apply to a single resource. For example, an application running in a multicontainer Docker environment on Elastic Beanstalk could have information about the Amazon EC2 instance, the Amazon ECS container running on the instance, and the Elastic Beanstalk environment itself.

`aws` (Segments)

All fields are optional.

- `account_id` – If your application sends segments to a different AWS account, record the ID of the account running your application.
- `cloudwatch_logs` – Array of objects that describe a single CloudWatch log group.

  - `log_group` – The CloudWatch Log Group name.
  - `arn` – The CloudWatch Log Group ARN.
- `ec2` – Information about an Amazon EC2 instance.
  - `instance_id` – The instance ID of the EC2 instance.
  - `instance_size` – The type of EC2 instance.
  - `ami_id` – The Amazon Machine Image ID.
  - `availability_zone` – The Availability Zone in which the instance is running.
- `ecs` – Information about an Amazon ECS container.
  - `container` – The hostname of your container.
  - `container_id` – The full container ID of your container.
  - `container_arn` – The ARN of your container instance.
- `eks` – Information about an Amazon EKS cluster.
  - pod – The hostname of your EKS pod.
  - `cluster_name` – The EKS cluster name.
  - `container_id` – The full container ID of your container.
- `elastic_beanstalk` – Information about an Elastic Beanstalk environment. You can find this information in a file named `/var/elasticbeanstalk/xray/environment.conf` on the latest Elastic Beanstalk platforms.
  - `environment_name` – The name of the environment.
  - `version_label` – The name of the application version that is currently deployed to the instance that served the request.
  - `deployment_id` – **number** indicating the ID of the last successful deployment to the instance that served the request.
- `xray` – Metadata about the type and version of instrumentation used.
  - `auto_instrumentation` – Boolean indicating whether auto-instrumentation was used (for example, the Java Agent).
  - `sdk_version` – The version of SDK or agent being used.
  - `sdk` – The type of SDK.

Example AWS block with plugins

```
"aws":{
   "elastic_beanstalk":{
      "version_label":"app-5a56-170119_190650-stage-170119_190650",
      "deployment_id":32,
      "environment_name":"scorekeep"
   },
   "ec2":{
      "availability_zone":"us-west-2c",
      "instance_id":"i-075ad396f12bc325a",
      "ami_id":
   },
   "cloudwatch_logs":[
      {
         "log_group":"my-cw-log-group",
         "arn":"arn:aws:logs:us-west-2:012345678912:log-group:my-cw-log-group"
      }
   ],
   "xray":{
      "auto_instrumentation":false,
      "sdk":"X-Ray for Java",
      "sdk_version":"2.8.0"
   }
}
```

For subsegments, record information about the AWS services and resources that your application accesses. X-Ray uses this information to create inferred segments that represent the downstream services in your service map.

aws (Subsegments)

All fields are optional.

- `operation` – The name of the API action invoked against an AWS service or resource.
- `account_id` – If your application accesses resources in a different account, or sends segments to a different account, record the ID of the account that owns the AWS resource that your application accessed.
- `region` – If the resource is in a region different from your application, record the region. For example, `us-west-2`.
- `request_id` – Unique identifier for the request.
- `queue_url` – For operations on an Amazon SQS queue, the queue's URL.
- `table_name` – For operations on a DynamoDB table, the name of the table.

Example Subsegment for a call to DynamoDB to save an item

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

Errors and exceptions

When an error occurs, you can record details about the error and exceptions that it generated. Record errors in segments when your application returns an error to the user, and in subsegments when a downstream call returns an error.

error types

Set one or more of the following fields to `true` to indicate that an error occurred. Multiple types can apply if errors compound. For example, a `429 Too Many Requests` error from a downstream call may cause your application to return `500 Internal Server Error`, in which case all three types would apply.

- `error` – **boolean** indicating that a client error occurred (response status code was 4XX Client Error).
- `throttle` – **boolean** indicating that a request was throttled (response status code was *429 Too Many Requests*).
- `fault` – **boolean** indicating that a server error occurred (response status code was 5XX Server Error).

Indicate the cause of the error by including a **cause** object in the segment or subsegment.

`cause`

A cause can be either a **16 character** exception ID or an object with the following fields:

- `working_directory` – The full path of the working directory when the exception occurred.
- `paths` – The **array** of paths to libraries or modules in use when the exception occurred.
- `exceptions` – The **array** of **exception** objects.

Include detailed information about the error in one or more **exception** objects.

`exception`

All fields are optional.

- `id` – A 64-bit identifier for the exception, unique among segments in the same trace, in **16 hexadecimal digits**.
- `message` – The exception message.
- `type` – The exception type.
- `remote` – **boolean** indicating that the exception was caused by an error returned by a downstream service.
- `truncated` – **integer** indicating the number of stack frames that are omitted from the `stack`.
- `skipped` – **integer** indicating the number of exceptions that were skipped between this exception and its child, that is, the exception that it caused.
- `cause` – Exception ID of the exception's parent, that is, the exception that caused this exception.
- `stack` – **array** of **stackFrame** objects.

If available, record information about the call stack in **stackFrame** objects.

`stackFrame`

All fields are optional.

- `path` – The relative path to the file.
- `line` – The line in the file.
- `label` – The function or method name.

SQL queries

You can create subsegments for queries that your application makes to an SQL database.

sql

All fields are optional.

- `connection_string` – For SQL Server or other database connections that don't use URL connection strings, record the connection string, excluding passwords.
- `url` – For a database connection that uses a URL connection string, record the URL, excluding passwords.
- `sanitized_query` – The database query, with any user provided values removed or replaced by a placeholder.
- `database_type` – The name of the database engine.
- `database_version` – The version number of the database engine.
- `driver_version` – The name and version number of the database engine driver that your application uses.
- `user` – The database username.
- `preparation` – `call` if the query used a `PreparedCall`; `statement` if the query used a `PreparedStatement`.

Example Subsegment with an SQL Query

```
{
  "id": "3fd8634e78ca9560",
  "start_time": 1484872218.696,
  "end_time": 1484872218.697,
  "name": "ebdb@aawijb5u25wdoy.cpamxznpdoq8.us-west-2.rds.amazonaws.com",
  "namespace": "remote",
  "sql" : {
    "url": "jdbc:postgresql://aawijb5u25wdoy.cpamxznpdoq8.us-
west-2.rds.amazonaws.com:5432/ebdb",
    "preparation": "statement",
    "database_type": "PostgreSQL",
    "database_version": "9.5.4",
    "driver_version": "PostgreSQL 9.4.1211.jre7",
    "user" : "dbuser",
    "sanitized_query" : "SELECT  *  FROM  customers  WHERE  customer_id=?;"
  }
}
```

AWS X-Ray concepts

AWS X-Ray receives data from services as *segments*. X-Ray then groups segments that have a common request into *traces*. X-Ray processes the traces to generate a *service graph* that provides a visual representation of your application.

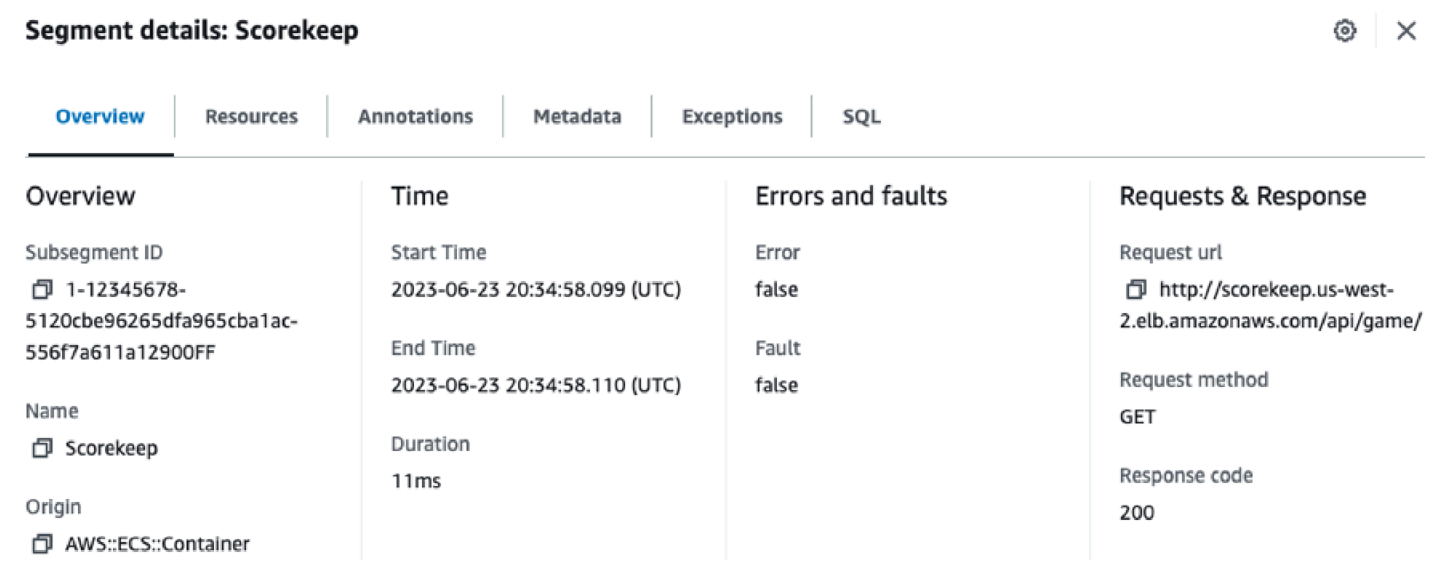
Concepts

- Segments
- Subsegments
- Service graph
- Traces
- Sampling
- Tracing header
- Filter expressions
- Groups
- Annotations and metadata
- Errors, faults, and exceptions

Segments

The compute resources running your application logic send data about their work as **segments**. A segment provides the resource's name, details about the request, and details about the work done. For example, when an HTTP request reaches your application, it can record the following data about:

- **The host** – hostname, alias or IP address
- **The request** – method, client address, path, user agent
- **The response** – status, content
- **The work done** – start and end times, subsegments
- **Issues that occur** – errors, faults and exceptions, including automatic capture of exception stacks.



Overview

Subsegment ID
1-12345678-5120cbe96265dfa965cba1ac-556f7a611a12900FF

Name
Scorekeep

Origin
AWS::ECS::Container

Time

Start Time
2023-06-23 20:34:58.099 (UTC)

End Time
2023-06-23 20:34:58.110 (UTC)

Duration
11ms

Errors and faults

Error
false

Fault
false

Requests & Response

Request url
http://scorekeep.us-west-2.elb.amazonaws.com/api/game/

Request method
GET

Response code
200

The X-Ray SDK gathers information from request and response headers, the code in your application, and metadata about the AWS resources on which it runs. You choose the data to collect by modifying your application configuration or code to instrument incoming requests, downstream requests, and AWS SDK clients.

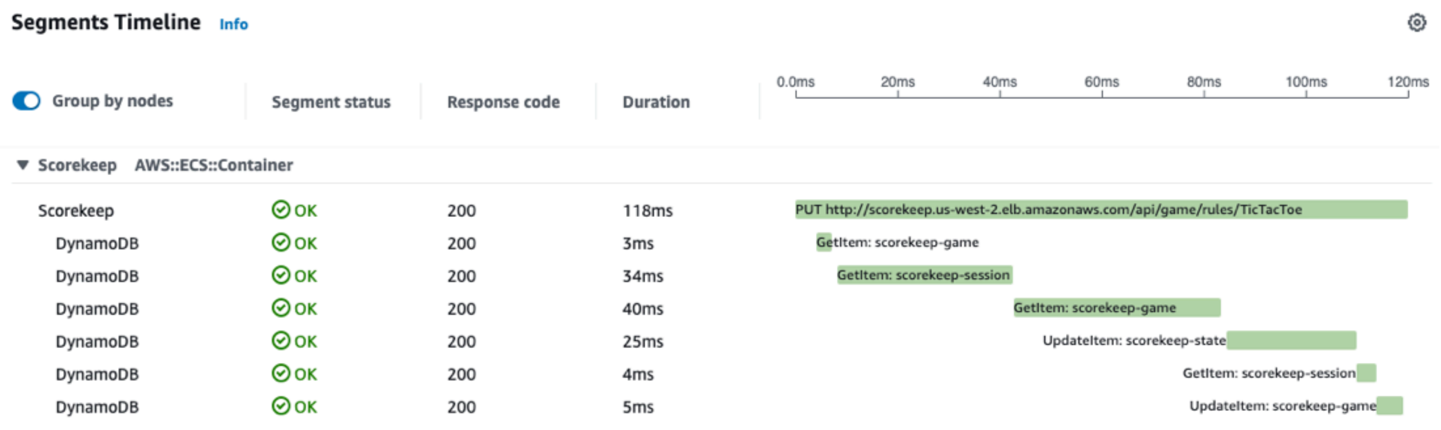
Forwarded Requests

If a load balancer or other intermediary forwards a request to your application, X-Ray takes the client IP from the `X-Forwarded-For` header in the request instead of from the source IP in the IP packet. The client IP that is recorded for a forwarded request can be forged, so it should not be trusted.

You can use the X-Ray SDK to record additional information such as annotations and metadata. For details about the structure and information that is recorded in segments and subsegments, see AWS X-Ray segment documents. Segment documents can be up to 64 kB in size.

Subsegments

A segment can break down the data about the work done into **subsegments**. Subsegments provide more granular timing information and details about downstream calls that your application made to fulfill the original request. A subsegment can contain additional details about a call to an AWS service, an external HTTP API, or an SQL database. You can even define arbitrary subsegments to instrument specific functions or lines of code in your application.



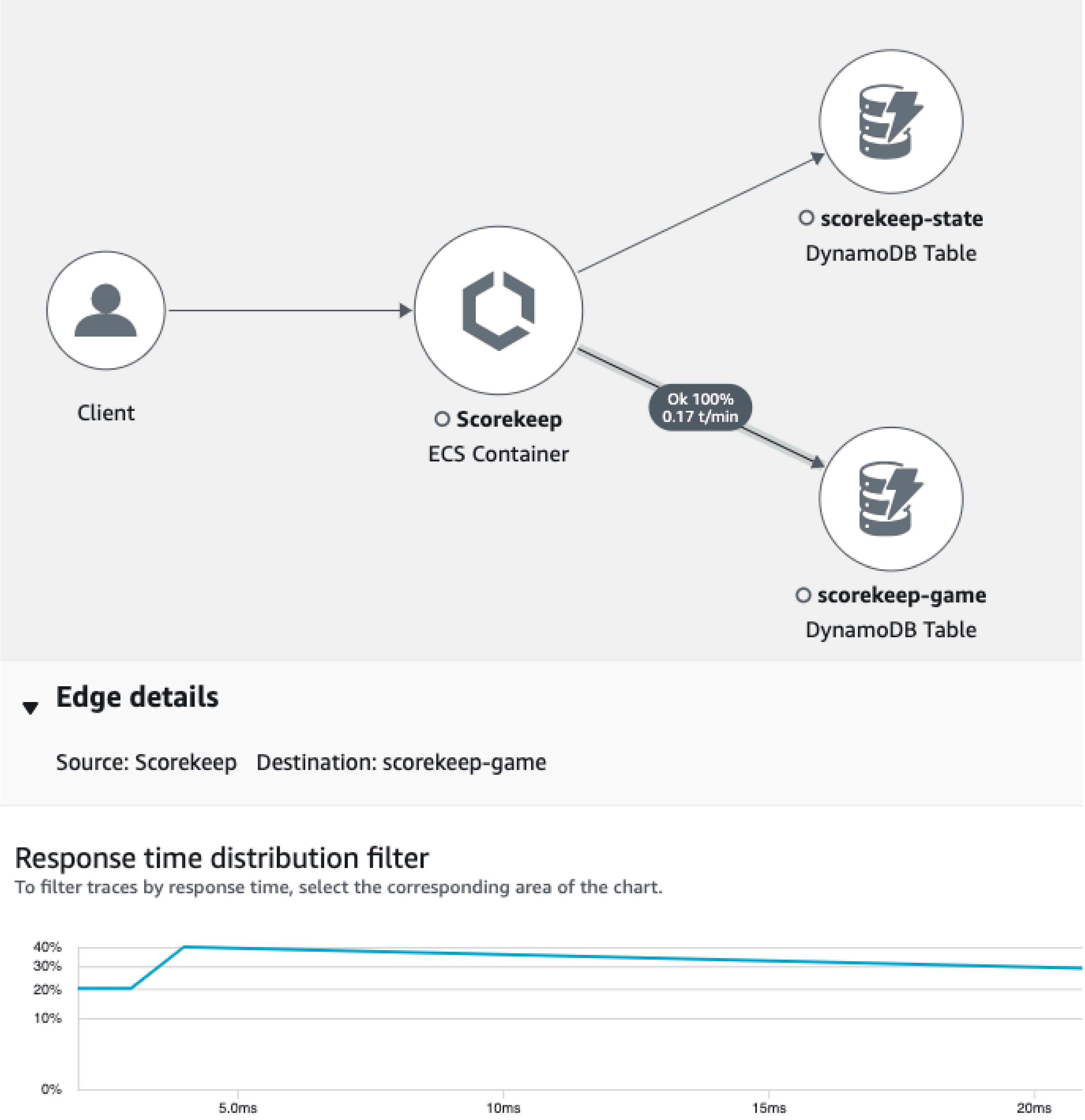
Segments Timeline Info

Group by nodes	Segment status	Response code	Duration	0.0ms 20ms 40ms 60ms 80ms 100ms 120ms
▼ Scorekeep AWS::ECS::Container				
Scorekeep	✓ OK	200	118ms	PUT http://scorekeep.us-west-2.elb.amazonaws.com/api/game/rules/TicTacToe
DynamoDB	✓ OK	200	3ms	GetItem: scorekeep-game
DynamoDB	✓ OK	200	34ms	GetItem: scorekeep-session
DynamoDB	✓ OK	200	40ms	GetItem: scorekeep-game
DynamoDB	✓ OK	200	25ms	UpdateItem: scorekeep-state
DynamoDB	✓ OK	200	4ms	GetItem: scorekeep-session
DynamoDB	✓ OK	200	5ms	UpdateItem: scorekeep-game

For services that don't send their own segments, like Amazon DynamoDB, X-Ray uses subsegments to generate *inferred segments* and downstream nodes on the trace map. This lets you see all of your downstream dependencies, even if they don't support tracing, or are external.

Subsegments represent your application's view of a downstream call as a client. If the downstream service is also instrumented, the segment that it sends replaces the inferred segment generated from the upstream client's subsegment. The node on the service graph always uses information from the service's segment, if it's available, while the edge between the two nodes uses the upstream service's subsegment.

For example, when you call DynamoDB with an instrumented AWS SDK client, the X-Ray SDK records a subsegment for that call. DynamoDB doesn't send a segment, so the inferred segment in the trace, the DynamoDB node on the service graph, and the edge between your service and DynamoDB all contain information from the subsegment.



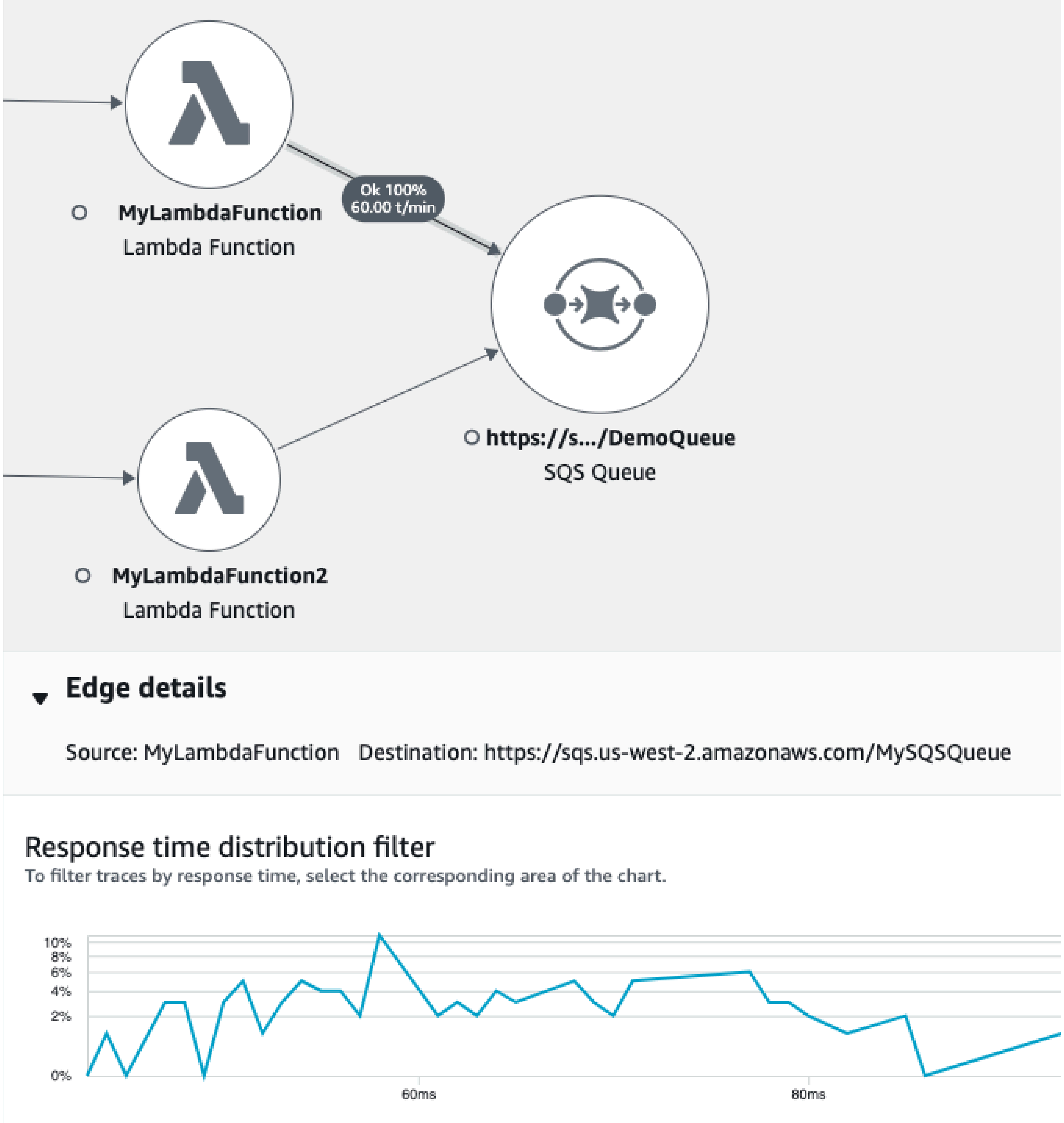
Edge details

Source: Scorekeep Destination: scorekeep-game

Response time distribution filter

To filter traces by response time, select the corresponding area of the chart.

When you call another instrumented service with an instrumented application, the downstream service sends its own segment to record its view of the same call that the upstream service recorded in a subsegment. In the service graph, both services' nodes contain timing and error information from those services' segments, while the edge between them contains information from the upstream service's subsegment.



▼ Edge details

Source: MyLambdaFunction Destination: https://sqs.us-west-2.amazonaws.com/MySQSQueue

Response time distribution filter

To filter traces by response time, select the corresponding area of the chart.

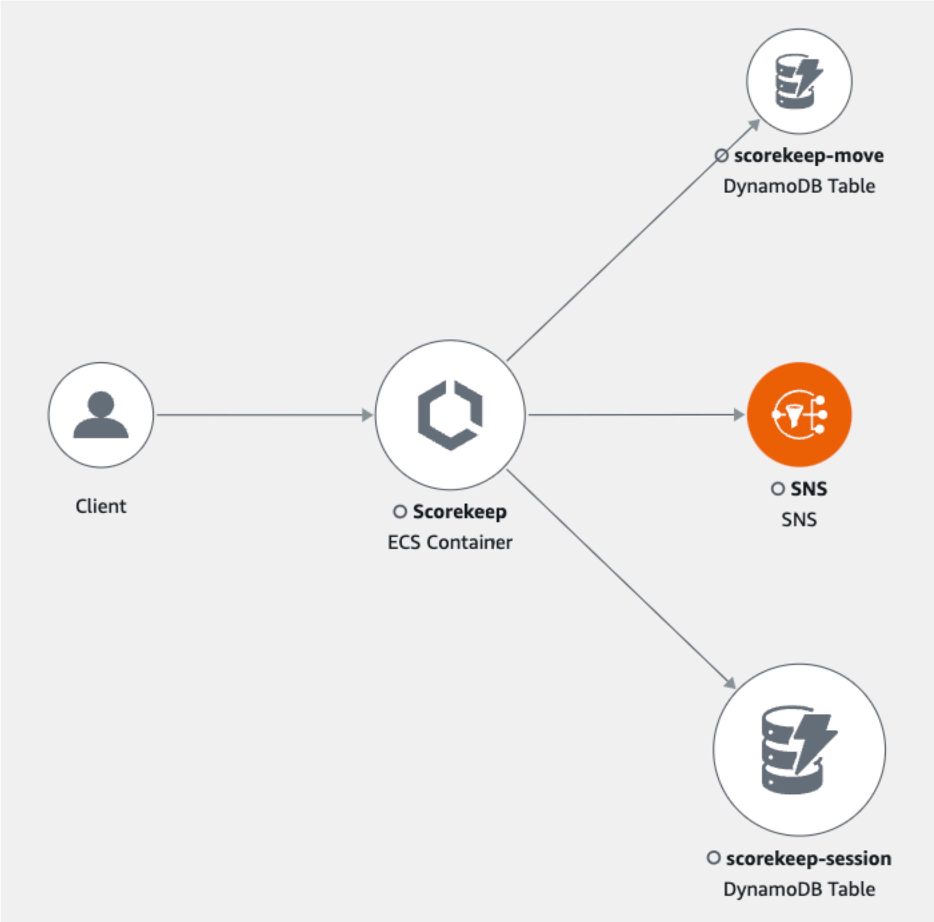
Both viewpoints are useful, as the downstream service records precisely when it started and ended work on the request, and the upstream service records the round trip latency, including time that the request spent traveling between the two services.

Service graph

X-Ray uses the data that your application sends to generate a **service graph**. Each AWS resource that sends data to X-Ray appears as a service in the graph. **Edges** connect the services that work together to serve requests. Edges connect clients to your application, and your application to the downstream services and resources that it uses.

> **Service Names**
>
> A segment's `name` should match the domain name or logical name of the service that generates the segment. However, this is not enforced. Any application that has permission to [`PutTraceSegments`](#) can send segments with any name.

A service graph is a JSON document that contains information about the services and resources that make up your application. The X-Ray console uses the service graph to generate a visualization or *service map*.



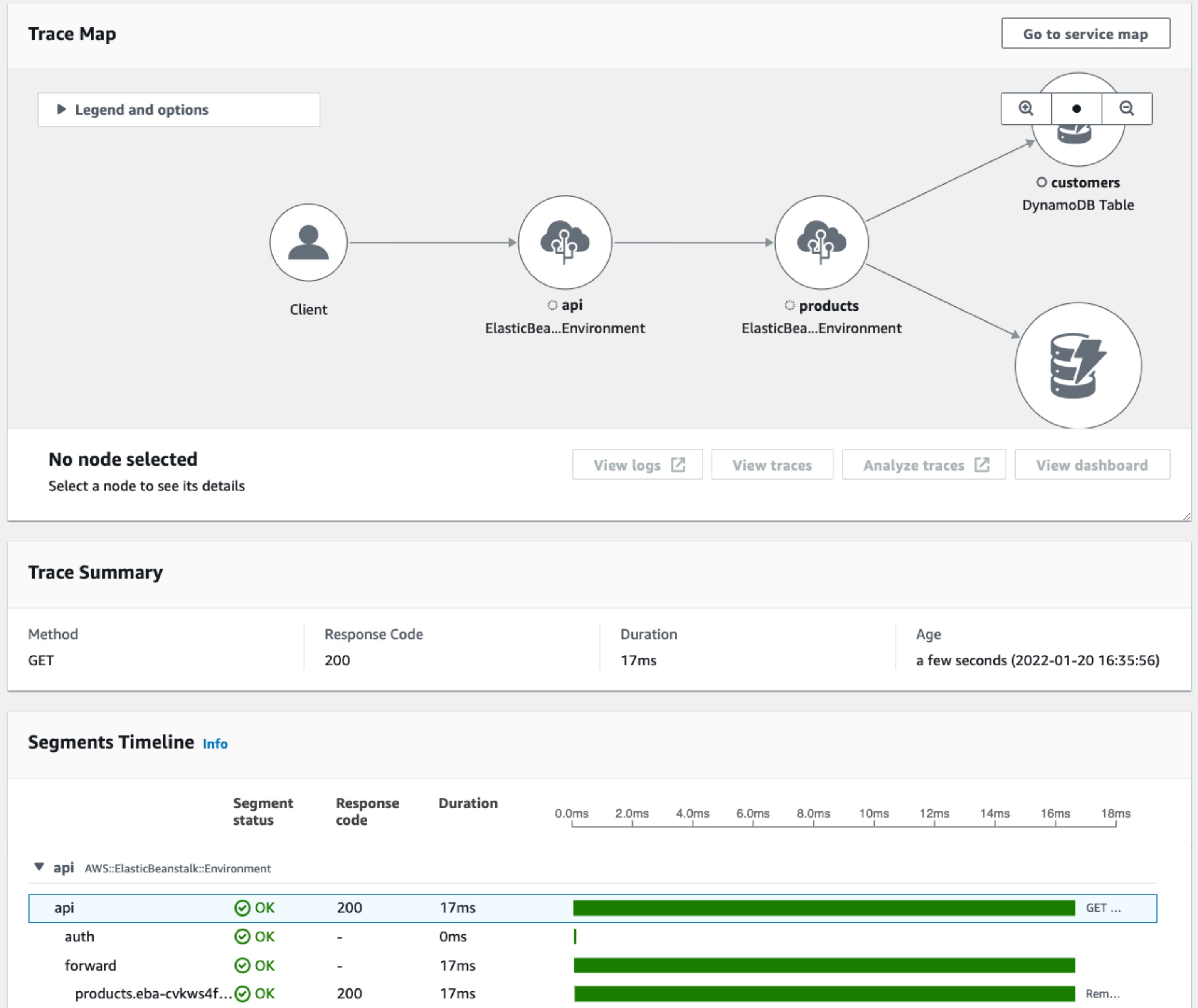
For a distributed application, X-Ray combines nodes from all services that process requests with the same trace ID into a single service graph. The first service that the request hits adds a tracing header that is propagated between the front end and services that it calls.

For example, Scorekeep runs a web API that calls a microservice (an AWS Lambda function) to generate a random name by using a Node.js library. The X-Ray SDK for Java generates the trace ID and includes it in calls to Lambda. Lambda sends tracing data and passes the trace ID to the function. The X-Ray SDK for Node.js also uses the trace ID to send data. As a result, nodes for the API, the Lambda service, and the Lambda function all appear as separate, but connected, nodes on the trace map.

Service graph data is retained for 30 days.

Traces

A **trace ID** tracks the path of a request through your application. A trace collects all the segments generated by a single request. That request is typically an HTTP GET or POST request that travels through a load balancer, hits your application code, and generates downstream calls to other AWS services or external web APIs. The first supported service that the HTTP request interacts with adds a trace ID header to the request, and propagates it downstream to track the latency, disposition, and other request data.



Trace Summary

Method	Response Code	Duration	Age
GET	200	17ms	a few seconds (2022-01-20 16:35:56)

Segments Timeline Info

▼ api AWS::ElasticBeanstalk::Environment

	Segment status	Response code	Duration	0.0ms – 18ms	
api	OK	200	17ms	🟩	GET ...
auth	OK	-	0ms		
forward	OK	-	17ms	🟩	
products.eba-cvkws4f...	OK	200	17ms	🟩	Rem...

See AWS X-Ray pricing for information about how X-Ray traces are billed. Trace data is retained for 30 days.

Sampling

To ensure efficient tracing and provide a representative sample of the requests that your application serves, the X-Ray SDK applies a **sampling** algorithm to determine which requests get traced. By default, the X-Ray SDK records the first request each second, and five percent of any additional requests.

To avoid incurring service charges when you are getting started, the default sampling rate is conservative. You can configure X-Ray to modify the default sampling rule and configure additional rules that apply sampling based on properties of the service or request.

For example, you might want to disable sampling and trace all requests for calls that modify state or handle users or transactions. For high-volume read-only calls, like background polling, health checks, or connection maintenance, you can sample at a low rate and still get enough data to see any issues that arise.

For more information, see Configuring sampling rules.

Tracing header

All requests are traced, up to a configurable minimum. After reaching that minimum, a percentage of requests are traced to avoid unnecessary cost. The sampling decision and trace ID are added to HTTP requests in **tracing headers** named `X-Amzn-Trace-Id`. The first X-Ray-integrated service that the request hits adds a tracing header, which is read by the X-Ray SDK and included in the response.

Example Tracing header with root trace ID and sampling decision

```
X-Amzn-Trace-Id: Root=1-5759e988-
bd862e3fe1be46a994272793;Parent=53995c3f42cd8ad8;Sampled=1
```

Tracing Header Security

A tracing header can originate from the X-Ray SDK, an AWS service, or the client request. Your application can remove `X-Amzn-Trace-Id` from incoming requests to avoid issues caused by users adding trace IDs or sampling decisions to their requests.

The tracing header can also contain a parent segment ID if the request originated from an instrumented application. For example, if your application calls a downstream HTTP web API with an instrumented HTTP client, the X-Ray SDK adds the segment ID for the original request to the tracing header of the downstream request. An instrumented application that serves the downstream request can record the parent segment ID to connect the two requests.

Example Tracing header with root trace ID, parent segment ID and sampling decision

```
X-Amzn-Trace-Id: Root=1-5759e988-
bd862e3fe1be46a994272793;Parent=53995c3f42cd8ad8;Sampled=1
```

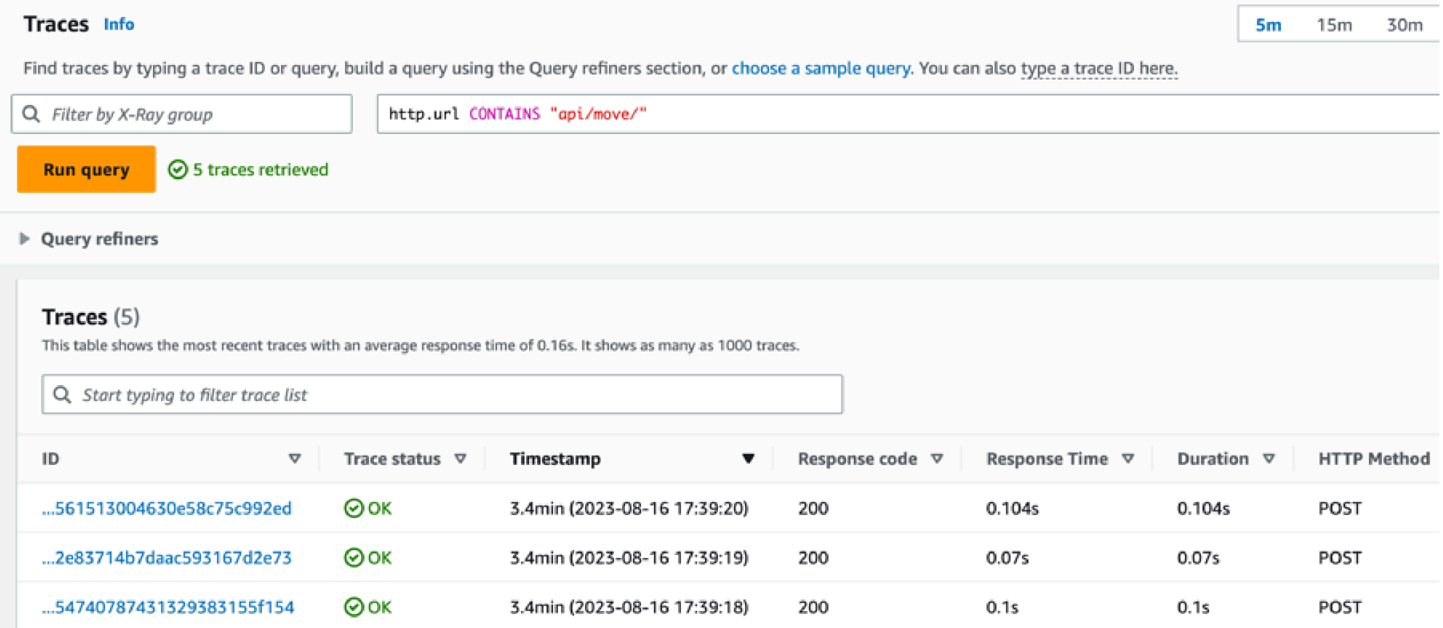
`Lineage` may be appended to the trace header by Lambda and other AWS services as part of their processing mechanisms, and should not be directly used.

Example Tracing header with Lineage

```
X-Amzn-Trace-Id: Root=1-5759e988-
bd862e3fe1be46a994272793;Parent=53995c3f42cd8ad8;Sampled=1;Lineage=25:a87bd80c:1
```

Filter expressions

Even with sampling, a complex application generates a lot of data. The AWS X-Ray console provides an easy-to-navigate view of the service graph. It shows health and performance information that helps you identify issues and opportunities for optimization in your application. For advanced tracing, you can drill down to traces for individual requests, or use **filter expressions** to find traces related to specific paths or users.



Find traces by typing a trace ID or query, build a query using the Query refiners section, or choose a sample query. You can also type a trace ID here.

`http.url CONTAINS "api/move/"`

5 traces retrieved

Traces (5)

This table shows the most recent traces with an average response time of 0.16s. It shows as many as 1000 traces.

ID	Trace status	Timestamp	Response code	Response Time	Duration	HTTP Method
...561513004630e58c75c992ed	OK	3.4min (2023-08-16 17:39:20)	200	0.104s	0.104s	POST
...2e83714b7daac593167d2e73	OK	3.4min (2023-08-16 17:39:19)	200	0.07s	0.07s	POST
...5474078743132938315 5f154	OK	3.4min (2023-08-16 17:39:18)	200	0.1s	0.1s	POST

Groups

Extending filter expressions, X-Ray also supports the group feature. Using a filter expression, you can define criteria by which to accept traces into the group.

You can call the group by name or by Amazon Resource Name (ARN) to generate its own service graph, trace summaries, and Amazon CloudWatch metrics. Once a group is created, incoming traces are checked against the group’s filter expression as they are stored in the X-Ray service. Metrics for the number of traces matching each criteria are published to CloudWatch every minute.

Updating a group's filter expression doesn't change data that's already recorded. The update applies only to subsequent traces. This can result in a merged graph of the new and old expressions. To avoid this, delete the current group and create a fresh one.

> **Note**
>
> Groups are billed by the number of retrieved traces that match the filter expression. For more information, see AWS X-Ray pricing.

For more information about groups, see Configuring groups.

Annotations and metadata

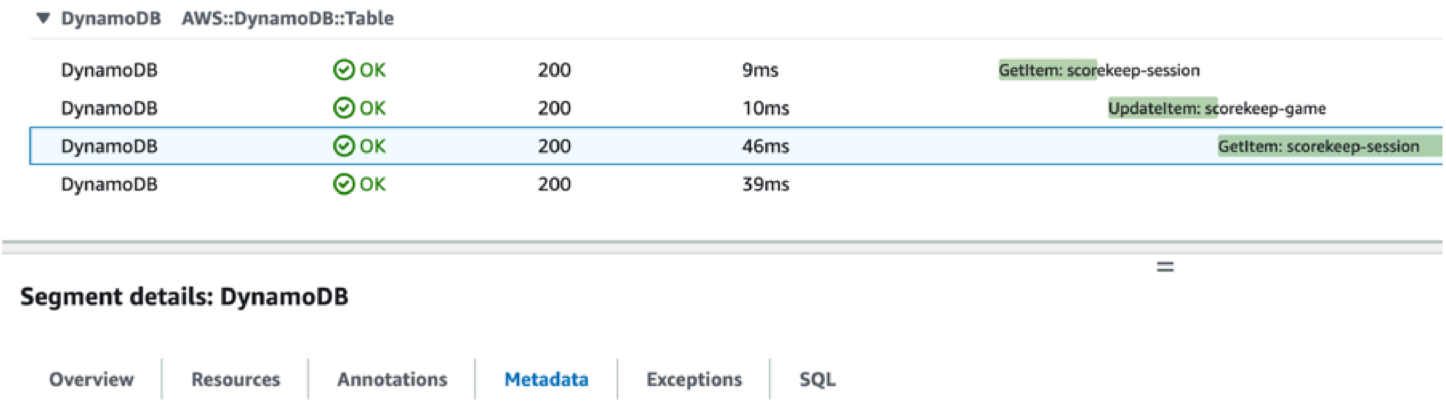
When you instrument your application, the X-Ray SDK records information about incoming and outgoing requests, the AWS resources used, and the application itself. You can add other information to the segment document as annotations and metadata. Annotations and metadata are aggregated at the trace level, and can be added to any segment or subsegment.

Annotations are simple key-value pairs that are indexed for use with filter expressions. Use annotations to record data that you want to use to group traces in the console, or when calling the `GetTraceSummaries` API.

X-Ray indexes up to 50 annotations per trace.

Metadata are key-value pairs with values of any type, including objects and lists, but that are not indexed. Use metadata to record data you want to store in the trace but don't need to use for searching traces.

You can view annotations and metadata in the segment or subsegment details window, within the Trace details page in the CloudWatch console.



▼ DynamoDB AWS::DynamoDB::Table

DynamoDB	✓ OK	200	9ms	GetItem: scorekeep-session
DynamoDB	✓ OK	200	10ms	UpdateItem: scorekeep-game
DynamoDB	✓ OK	200	46ms	GetItem: scorekeep-session
DynamoDB	✓ OK	200	39ms	

Segment details: DynamoDB

Errors, faults, and exceptions

X-Ray tracks errors that occur in your application code, and errors that are returned by downstream services. Errors are categorized as follows.

- **`Error`** – Client errors (400 series errors)
- **`Fault`** – Server faults (500 series errors)
- **`Throttle`** – Throttling errors (429 Too Many Requests)

When an exception occurs while your application is serving an instrumented request, the X-Ray SDK records details about the exception, including the stack trace, if available. You can view exceptions under segment details in the X-Ray console.

Security in AWS X-Ray

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. The effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS compliance programs. To learn about the compliance programs that apply to X-Ray, 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 organization's requirements, and applicable laws and regulations.

This documentation will help you understand how to apply the shared responsibility model when using X-Ray. The following topics show you how to configure X-Ray to meet your security and compliance objectives. You'll also learn how to use other AWS services that can help you to monitor and secure your X-Ray resources.

Topics

- Data protection in AWS X-Ray
- Identity and access management for AWS X-Ray
- Compliance validation for AWS X-Ray
- Resilience in AWS X-Ray
- Infrastructure security in AWS X-Ray

Data protection in AWS X-Ray

AWS X-Ray always encrypts traces and related data at rest. When you need to audit and disable encryption keys for compliance or internal requirements, you can configure X-Ray to use an AWS Key Management Service (AWS KMS) key to encrypt data.

X-Ray provides an AWS managed key named `aws/xray`. Use this key when you just want to audit key usage in AWS CloudTrail and don't need to manage the key itself. When you need to manage access to the key or configure key rotation, you can create a customer managed key.

When you change encryption settings, X-Ray spends some time generating and propagating data keys. While the new key is being processed, X-Ray may encrypt data with a combination of the new and old settings. Existing data is not re-encrypted when you change encryption settings.

> **Note**
>
> AWS KMS charges when X-Ray uses a KMS key to encrypt or decrypt trace data.
>
> - **Default encryption** – Free.
> - **AWS managed key** – Pay for key use.
> - **customer managed key** – Pay for key storage and use.
>
> See AWS Key Management Service Pricing for details.

> **Note**
>
> X-Ray insights notifications sends events to Amazon EventBridge, which does not currently support customer managed keys. For more information, see Data Protection in Amazon EventBridge.

You must have user-level access to a customer managed key to configure X-Ray to use it, and to then view encrypted traces. See User permissions for encryption for more information.

CloudWatch console

To configure X-Ray to use a KMS key for encryption using the CloudWatch console

1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Choose **Settings** in the left navigation pane.
3. Choose **View settings** under **Encryption** within the **X-Ray traces** section.
4. Choose **Edit** in the **Encryption configuration** section.

5. Choose **Use a KMS key**.
6. Choose a key from the dropdown menu:
   - **aws/xray** – Use the AWS managed key.
   - *key alias* – Use a customer managed key in your account.
   - **Manually enter a key ARN** – Use a customer managed key in a different account. Enter the full Amazon Resource Name (ARN) of the key in the field that appears.
7. Choose **Update encryption**.

X-Ray console

To configure X-Ray to use a KMS key for encryption using the X-Ray console

1. Open the X-Ray console.
2. Choose **Encryption**.
3. Choose **Use a KMS key**.
4. Choose a key from the dropdown menu:
   - **aws/xray** – Use the AWS managed key.
   - *key alias* – Use a customer managed key in your account.
   - **Manually enter a key ARN** – Use a customer managed key in a different account. Enter the full Amazon Resource Name (ARN) of the key in the field that appears.
5. Choose **Apply**.

> **Note**
>
> X-Ray does not support asymmetric KMS keys.

If X-Ray is unable to access your encryption key, it stops storing data. This can happen if your user loses access to the KMS key, or if you disable a key that's currently in use. When this happens, X-Ray shows a notification in the navigation bar.

To configure encryption settings with the X-Ray API, see Configuring sampling, groups, and encryption settings with the AWS X-Ray API.

Identity and access management for AWS X-Ray

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 X-Ray 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 X-Ray works with IAM
- AWS X-Ray identity-based policy examples
- Troubleshooting AWS X-Ray identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in X-Ray.

Service user – If you use the X-Ray service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more X-Ray 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 X-Ray, see Troubleshooting AWS X-Ray identity and access.

Service administrator – If you're in charge of X-Ray resources at your company, you probably have full access to X-Ray. It's your job to determine which X-Ray 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 X-Ray, see How AWS X-Ray 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 X-Ray. To view example X-Ray identity-based policies that you can use in IAM, see AWS X-Ray 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 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 AWS Signature Version 4 for API requests 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 in the *AWS IAM Identity Center User Guide* and AWS Multi-factor authentication in IAM 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 in the *IAM User Guide*.

IAM 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 Use cases for IAM users 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. To temporarily assume an IAM role in the AWS Management Console, you can switch from a user to an IAM role (console). 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 Methods to assume a role 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 Create a role for a third-party identity provider (federation) 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 Cross account resource access in IAM 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 Create 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 Use an IAM role to grant permissions to applications running on Amazon EC2 instances 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 Define custom IAM permissions with customer managed 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 Choose 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 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 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 Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – RCPs are JSON policies that you can use to set the maximum available permissions for resources in your accounts without updating the IAM policies attached

to each resource that you own. The RCP limits permissions for resources in member accounts and can impact the effective permissions for identities, including the AWS account root user, regardless of whether they belong to your organization. For more information about Organizations and RCPs, including a list of AWS services that support RCPs, see Resource control policies (RCPs) 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 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 X-Ray works with IAM

Before you use IAM to manage access to X-Ray, you should understand what IAM features are available to use with X-Ray. To get a high-level view of how X-Ray and other AWS services work with IAM, see AWS services That Work with IAM in the *IAM User Guide*.

You can use AWS Identity and Access Management (IAM) to grant X-Ray permissions to users and compute resources in your account. IAM controls access to the X-Ray service at the API level to enforce permissions uniformly, regardless of which client (console, AWS SDK, AWS CLI) your users employ.

To use the X-Ray console to view trace maps and segments, you only need read permissions. To enable console access, add the `AWSXrayReadOnlyAccess` managed policy to your IAM user.

For local development and testing, create an IAM role with read and write permissions. Assume the role and store temporary credentials for the role. You can use these credentials with the X-Ray daemon, the AWS CLI, and the AWS SDK. See using temporary security credentials with the AWS CLI for more information.

To deploy your instrumented app to AWS, create an IAM role with write permissions and assign it to the resources running your application. `AWSXRayDaemonWriteAccess` includes permission to upload traces, and some read permissions as well to support the use of sampling rules.

The read and write policies do not include permission to configure encryption key settings and sampling rules. Use `AWSXrayFullAccess` to access these settings, or add configuration APIs in a custom policy. For encryption and decryption with a customer managed key that you create, you also need permission to use the key.

Topics

- X-Ray identity-based policies
- X-Ray resource-based policies
- Authorization based on X-Ray tags
- Running your application locally
- Running your application in AWS
- User permissions for encryption

X-Ray identity-based policies

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. X-Ray supports specific actions, resources, and condition keys. To learn about all of the elements that you use in a JSON policy, see IAM JSON Policy Elements Reference in the *IAM User Guide*.

Actions

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**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

Policy actions in X-Ray use the following prefix before the action: `xray:`. For example, to grant someone permission to retrieve group resource details with the X-Ray `GetGroup` API operation, you include the `xray:GetGroup` action in their policy. Policy statements must include either an `Action` or `NotAction` element. X-Ray defines its own set of actions that describe tasks that you can perform with this service.

To specify multiple actions in a single statement, separate them with commas as follows:

```
"Action": [
      "xray:action1",
      "xray:action2"
```

You can specify multiple actions using wildcards (*). For example, to specify all actions that begin with the word `Get`, include the following action:

```
"Action": "xray:Get*"
```

To see a list of X-Ray actions, see Actions Defined by AWS X-Ray in the *IAM User Guide*.

Resources

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**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. As a best practice, specify a resource using its Amazon Resource Name (ARN). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

You can control access to resources by using an IAM policy. For actions that support resource-level permissions, you use an Amazon Resource Name (ARN) to identify the resource that the policy applies to.

All X-Ray actions can be used in an IAM policy to grant or deny users permission to use that action. However, not all X-Ray actions support resource-level permissions, which enable you to specify the resources on which an action can be performed.

For actions that don't support resource-level permissions, you must use "*" as the resource.

The following X-Ray actions support resource-level permissions:

- `CreateGroup`
- `GetGroup`
- `UpdateGroup`
- `DeleteGroup`
- `CreateSamplingRule`
- `UpdateSamplingRule`
- `DeleteSamplingRule`

The following is an example of an identity-based permissions policy for a `CreateGroup` action. The example shows the use of an ARN relating to Group name `local-users` with the unique ID as a wildcard. The unique ID is generated when the group is created, and so it can't be predicted in the policy in advance. When using `GetGroup`, `UpdateGroup`, or `DeleteGroup`, you can define this as either a wildcard or the exact ARN, including ID.

> **Note**
>
> The ARN of a sampling rule is defined by its name. Unlike group ARNs, sampling rules have no uniquely generated ID.

To see a list of X-Ray resource types and their ARNs, see Resources Defined by AWS X-Ray in the *IAM User Guide*. To learn with which actions you can specify the ARN of each resource, see Actions Defined by AWS X-Ray.

Condition keys

X-Ray does not provide any service-specific condition keys, but it does support using some global condition keys. To see all AWS global condition keys, see AWS Global Condition Context Keys in the *IAM User Guide*.

Examples

To view examples of X-Ray identity-based policies, see AWS X-Ray identity-based policy examples.

X-Ray resource-based policies

X-Ray supports resource-based policies for current and future AWS service integration, such as Amazon SNS active tracing. X-Ray resource-based policies can be updated by other AWS

Management Consoles, or through the AWS SDK or CLI. For example, the Amazon SNS console attempts to automatically configure resource-based policy for sending traces to X-Ray. The following policy document provides an example of manually configuring X-Ray resource-based policy.

Example Example X-Ray resource-based policy for Amazon SNS active tracing

This example policy document specifies the permissions that Amazon SNS needs to send trace data to X-Ray:

```
{
    Version: "2012-10-17",
    Statement: [
      {
        Sid: "SNSAccess",
        Effect: Allow,
        Principal: {
          Service: "sns.amazonaws.com",
        },
        Action: [
          "xray:PutTraceSegments",
          "xray:GetSamplingRules",
          "xray:GetSamplingTargets"
        ],
        Resource: "*",
        Condition: {
          StringEquals: {
            "aws:SourceAccount": "account-id"
          },
          StringLike: {
            "aws:SourceArn": "arn:partition:sns:region:account-id:topic-name"
          }
        }
      }
    ]
  }
```

Use the CLI to create a resource-based policy that gives Amazon SNS permissions to send trace data to X-Ray:

```
aws xray put-resource-policy --policy-name MyResourcePolicy --policy-document
 '{ "Version": "2012-10-17", "Statement": [ { "Sid": "SNSAccess", "Effect": "Allow",
 "Principal": { "Service": "sns.amazonaws.com" }, "Action": [ "xray:PutTraceSegments",
 "xray:GetSamplingRules", "xray:GetSamplingTargets" ], "Resource": "*",
 "Condition": { "StringEquals": { "aws:SourceAccount": "account-id" }, "StringLike":
 { "aws:SourceArn": "arn:partition:sns:region:account-id:topic-name" } } } ] }'
```

To use these examples, replace `partition`, `region`, `account-id`, and `topic-name` with your specific AWS partition, region, account ID, and Amazon SNS topic name. To give all Amazon SNS topics permission to send trace data to X-Ray, replace the topic name with `*`.

Authorization based on X-Ray tags

You can attach tags to X-Ray groups or sampling rules, or pass tags in a request to X-Ray. To control access based on tags, you provide tag information in the condition element of a policy using the `xray:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys. For more information about tagging X-Ray resources, see Tagging X-Ray sampling rules and groups.

To view an example identity-based policy for limiting access to a resource based on the tags on that resource, see Managing access to X-Ray groups and sampling rules based on tags.

Running your application locally

Your instrumented application sends trace data to the X-Ray daemon. The daemon buffers segment documents and uploads them to the X-Ray service in batches. The daemon needs write permissions to upload trace data and telemetry to the X-Ray service.

When you run the daemon locally, create an IAM role, assume the role and store temporary credentials in environment variables, or in a file named `credentials` within a folder named `.aws` in your user folder. See using temporary security credentials with the AWS CLI for more information.

Example ~/.aws/credentials

```
[default]
aws_access_key_id={access key ID}
aws_secret_access_key={access key}
aws_session_token={AWS session token}
```

If you already configured credentials for use with the AWS SDK or AWS CLI, the daemon can use those. If multiple profiles are available, the daemon uses the default profile.

Running your application in AWS

When you run your application on AWS, use a role to grant permission to the Amazon EC2 instance or Lambda function that runs the daemon.

- **Amazon Elastic Compute Cloud (Amazon EC2)** – Create an IAM role and attach it to the EC2 instance as an instance profile.
- **Amazon Elastic Container Service (Amazon ECS)** – Create an IAM role and attach it to container instances as a container instance IAM role.
- **AWS Elastic Beanstalk (Elastic Beanstalk)** – Elastic Beanstalk includes X-Ray permissions in its default instance profile. You can use the default instance profile, or add write permissions to a custom instance profile.
- **AWS Lambda (Lambda)** – Add write permissions to your function's execution role.

To create a role for use with X-Ray

1. Open the IAM console.
2. Choose **Roles**.
3. Choose **Create New Role**.
4. For **Role Name**, type **`xray-application`**. Choose **Next Step.**
5. For **Role Type**, choose **Amazon EC2**.
6. Attach the following managed policy to give your application access to AWS services:

   - **AWSXRayDaemonWriteAccess** – Gives the X-Ray daemon permission to upload trace data.

   If your application uses the AWS SDK to access other services, add policies that grant access to those services.
7. Choose **Next Step**.
8. Choose **Create Role**.

User permissions for encryption

X-Ray encrypts all trace data and by default, and you can configure it to use a key that you manage. If you choose a AWS Key Management Service customer managed key, you need to ensure that the key's access policy lets you grant permission to X-Ray to use it to encrypt. Other users in your account also need access to the key to view encrypted trace data in the X-Ray console.

For a customer managed key, configure your key with an access policy that allows the following actions:

- User who configures the key in X-Ray has permission to call `kms:CreateGrant` and `kms:DescribeKey`.
- Users who can access encrypted trace data have permission to call `kms:Decrypt`.

When you add a user to the **Key users** group in the key configuration section of the IAM console, they have permission for both of these operations. Permission only needs to be set on the key policy, so you don't need any AWS KMS permissions on your users, groups, or roles. For more information, see Using Key Policies in the AWS KMS Developer Guide.

For default encryption, or if you choose the AWS managed CMK (`aws/xray`), permission is based on who has access to X-Ray APIs. Anyone with access to `PutEncryptionConfig`, included in `AWSXrayFullAccess`, can change the encryption configuration. To prevent a user from changing the encryption key, do not give them permission to use `PutEncryptionConfig`.

AWS X-Ray identity-based policy examples

By default, users and roles don't have permission to create or modify X-Ray 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 X-Ray console
- Allow users to view their own permissions
- Managing access to X-Ray groups and sampling rules based on tags
- IAM managed policies for X-Ray

- X-Ray updates to AWS managed policies
- Specifying a resource within an IAM policy

Policy best practices

Identity-based policies determine whether someone can create, access, or delete X-Ray 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 Validate policies with IAM Access Analyzer 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 Secure API access with MFA 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 X-Ray console

To access the AWS X-Ray console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the X-Ray 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.

To ensure that those entities can still use the X-Ray console, attach the `AWSXRayReadOnlyAccess` AWS managed policy to the entities. This policy is described in more detail in IAM managed policies for X-Ray. For more information, see Adding Permissions to a User in the *IAM User Guide*.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that you're trying to perform.

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": "*"
        }
    ]
}
```

Managing access to X-Ray groups and sampling rules based on tags

You can use conditions in your identity-based policy to control access to X-Ray groups and sampling rules based on tags. The following example policy could be used to deny a user role the permissions to create, delete, or update groups with the tags `stage:prod` or `stage:preprod`. For more information about tagging X-Ray sampling rules and groups, see [Tagging X-Ray sampling rules and groups](#).

To deny the creation of a sampling rule, use `aws:RequestTag` to indicate tags that cannot be passed as part of a creation request. To deny the update or deletion of a sampling rule, use `aws:ResourceTag` to deny actions based on the tags on those resources.

You can attach these policies (or combine them into a single policy, then attach the policy) to the users in your account. For the user to make changes to a group or sampling rule, the group or sampling rule must not be tagged `stage=prepod` or `stage=prod`. The condition tag key `Stage` matches both `Stage` and `stage` because condition key names are not case-sensitive. For more information about the condition block, see [IAM JSON Policy Elements: Condition](#) in the *IAM User Guide*.

A user with a role that has the following policy attached cannot add the tag `role:admin` to resources, and cannot remove tags from a resource that has `role:admin` associated with it.

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowAllXRay",
            "Effect": "Allow",
            "Action": "xray:*",
            "Resource": "*"
        },
        {
            "Sid": "DenyRequestTagAdmin",
            "Effect": "Deny",
            "Action": "xray:TagResource",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/role": "admin"
                }
            }
        },
        {
            "Sid": "DenyResourceTagAdmin",
            "Effect": "Deny",
            "Action": "xray:UntagResource",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/role": "admin"
                }
            }
        }
    ]
}
```

IAM managed policies for X-Ray

To make granting permissions easy, IAM supports **managed policies** for each service. A service can update these managed policies with new permissions when it releases new APIs. AWS X-Ray provides managed policies for read only, write only, and administrator use cases.

- `AWSXrayReadOnlyAccess` – Read permissions for using the X-Ray console, AWS CLI, or AWS SDK to get trace data, trace maps, insights, and X-Ray configuration from the X-Ray API. Includes Observability Access Manager (OAM) `oam:ListSinks` and `oam:ListAttachedSinks` permissions to allow the console to view traces shared from source accounts as part of CloudWatch cross-account observability. The `BatchGetTraceSummaryById` and `GetDistinctTraceGraphs` API actions are not intended to be called by your code, and not included in the AWS CLI and AWS SDKs.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:GetSamplingRules",
                "xray:GetSamplingTargets",
                "xray:GetSamplingStatisticSummaries",
                "xray:BatchGetTraces",
                "xray:BatchGetTraceSummaryById",
                "xray:GetDistinctTraceGraphs",
                "xray:GetServiceGraph",
                "xray:GetTraceGraph",
                "xray:GetTraceSummaries",
                "xray:GetGroups",
                "xray:GetGroup",
                "xray:ListTagsForResource",
                "xray:ListResourcePolicies",
                "xray:GetTimeSeriesServiceStatistics",
                "xray:GetInsightSummaries",
                "xray:GetInsight",
                "xray:GetInsightEvents",
                "xray:GetInsightImpactGraph",
                "oam:ListSinks"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "oam:ListAttachedLinks"
            ],
            "Resource": "arn:aws:oam:*:*:sink/*"
        }
}
```

- `AWSXRayDaemonWriteAccess` – Write permissions for using the X-Ray daemon, AWS CLI, or AWS SDK to upload segment documents and telemetry to the X-Ray API. Includes read permissions to get sampling rules and report sampling results.

  JSON

  ```
  {
      "Version": "2012-10-17",
      "Statement": [
          {
              "Effect": "Allow",
              "Action": [
                  "xray:PutTraceSegments",
                  "xray:PutTelemetryRecords",
                  "xray:GetSamplingRules",
                  "xray:GetSamplingTargets",
                  "xray:GetSamplingStatisticSummaries"
              ],
              "Resource": [
                  "*"
              ]
          }
      ]
  }
  ```

- `AWSXrayCrossAccountSharingConfiguration` – Grants permissions to create, manage, and view Observability Access Manager links for sharing X-Ray resources between accounts. Used to enable CloudWatch cross-account observability between source and monitoring accounts.

  JSON

  ```
  {
      "Version": "2012-10-17",
      "Statement": [
          {
              "Effect": "Allow",
              "Action": [
                  "xray:Link",
                  "oam:ListLinks"
              ],
              "Resource": "*"
  ```

```
        },
        {
            "Effect": "Allow",
            "Action": [
                "oam:DeleteLink",
                "oam:GetLink",
                "oam:TagResource"
            ],
            "Resource": "arn:aws:oam:*:*:link/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "oam:CreateLink",
                "oam:UpdateLink"
            ],
            "Resource": [
                "arn:aws:oam:*:*:link/*",
                "arn:aws:oam:*:*:sink/*"
            ]
        }
    ]

}
```

- `AWSXrayFullAccess` – Permission to use all X-Ray APIs, including read permissions, write permissions, and permission to configure encryption key settings and sampling rules. Includes Observability Access Manager (OAM) `oam:ListSinks` and `oam:ListAttachedSinks` permissions to allow the console to view traces shared from source accounts as part of CloudWatch cross-account observability.

  JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:*",
                "oam:ListSinks"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "oam:ListAttachedLinks"
            ],
            "Resource": "arn:aws:oam:*:*:sink/*"
        }
    ]
}
```

To add a managed policy to an IAM user, group, or role

1. Open the IAM console.
2. Open the role associated with your instance profile, an IAM user, or an IAM group.
3. Under **Permissions**, attach the managed policy.

X-Ray updates to AWS managed policies

View details about updates to AWS managed policies for X-Ray since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the X-Ray Document history page.

Change	Description	Date
IAM managed policies for X-Ray – Added new `AWSXrayCrossAccountSharingConfiguration` , and updated `AWSXrayReadOnlyAccess` and `AWSXrayFullAccess` policies.	X-Ray added Observability Access Manager (OAM) permissions `oam:ListSinks` and `oam:ListAttachedSinks` to these policies to allow the console to view traces shared from source accounts as part of CloudWatch cross-account observability.	November 27, 2022

Change	Description	Date
IAM managed policies for X-Ray – Update to `AWSXrayReadOnlyAccess` policy.	X-Ray added an API action, `ListResourcePolicies` .	November 15, 2022
Using the X-Ray console – Update to `AWSXrayReadOnlyAccess` policy	X-Ray added two new API actions, `BatchGetTraceSummaryById` and `GetDistinctTraceGraphs` . These actions are not intended to be called by your code. Therefore, these API actions are not included in the AWS CLI and AWS SDKs.	November 11, 2022

Specifying a resource within an IAM policy

You can control access to resources by using an IAM policy. For actions that support resource-level permissions, you use an Amazon Resource Name (ARN) to identify the resource that the policy applies to.

All X-Ray actions can be used in an IAM policy to grant or deny users permission to use that action. However, not all X-Ray actions support resource-level permissions, which enable you to specify the resources on which an action can be performed.

For actions that don't support resource-level permissions, you must use "*" as the resource.

The following X-Ray actions support resource-level permissions:

- `CreateGroup`
- `GetGroup`
- `UpdateGroup`
- `DeleteGroup`
- `CreateSamplingRule`

- `UpdateSamplingRule`
- `DeleteSamplingRule`

The following is an example of an identity-based permissions policy for a `CreateGroup` action. The example shows the use of an ARN relating to Group name `local-users` with the unique ID as a wildcard. The unique ID is generated when the group is created, and so it can't be predicted in the policy in advance. When using `GetGroup`, `UpdateGroup`, or `DeleteGroup`, you can define this as either a wildcard or the exact ARN, including ID.

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:CreateGroup"
            ],
            "Resource": [
                "arn:aws:xray:eu-west-1:123456789012:group/local-users/*"
            ]
        }
    ]
}
```

The following is an example of an identity-based permissions policy for a `CreateSamplingRule` action.

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:CreateSamplingRule"
            ],
            "Resource": [
                "arn:aws:xray:eu-west-1:123456789012:sampling-rule/base-
scorekeep"
            ]
        }
    ]
}
```

> **Note**
>
> The ARN of a sampling rule is defined by its name. Unlike group ARNs, sampling rules have no uniquely generated ID.

Troubleshooting AWS X-Ray identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with X-Ray and IAM.

Topics

- I Am not authorized to perform an action in X-Ray
- I Am not authorized to perform iam:PassRole
- I'm an administrator and want to allow others to access X-Ray
- I want to allow people outside of my AWS account to access my X-Ray resources

I Am not authorized to perform an action in X-Ray

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 sampling rule but does not have `xray:GetSamplingRules` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to
 perform: xray:GetSamplingRules on resource: arn:${Partition}:xray:${Region}:
${Account}:sampling-rule/${SamplingRuleName}
```

In this case, Mateo asks his administrator to update his policies to allow him to access the sampling rule resource using the `xray:GetSamplingRules` 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 X-Ray.

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 X-Ray. 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.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

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 allow others to access X-Ray

To allow others to access X-Ray, you must grant permission to the people or applications that need access. If you are using AWS IAM Identity Center to manage people and applications, you assign permission sets to users or groups to define their level of access. Permission sets automatically create and assign IAM policies to IAM roles that are associated with the person or application. For more information, see Permission sets in the *AWS IAM Identity Center User Guide*.

If you are not using IAM Identity Center, you must create IAM entities (users or roles) for the people or applications that need access. You must then attach a policy to the entity that grants them the correct permissions in X-Ray. After the permissions are granted, provide the credentials to the user or application developer. They will use those credentials to access AWS. To learn more about creating IAM users, groups, policies, and permissions, see IAM Identities and Policies and permissions in IAM in the *IAM User Guide*.

I want to allow people outside of my AWS account to access my X-Ray 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 X-Ray supports these features, see How AWS X-Ray 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 Cross account resource access in IAM in the *IAM User Guide*.

Logging and monitoring in AWS X-Ray

Monitoring is an important part of maintaining the reliability, availability, and performance of your AWS solutions. You should collect monitoring data from all of the parts of your AWS solution so that you can more easily debug a multi-point failure if one occurs. AWS provides several tools for monitoring your X-Ray resources and responding to potential incidents:

AWS CloudTrail Logs

> AWS X-Ray integrates with AWS CloudTrail to record API actions made by a user, a role, or an AWS service in X-Ray. You can use CloudTrail to monitor X-Ray API requests in real time and store logs in Amazon S3, Amazon CloudWatch Logs, and Amazon CloudWatch Events. For more information, see Logging X-Ray API calls with AWS CloudTrail.

AWS Config Tracking

AWS X-Ray integrates with AWS Config to record configuration changes made to your X-Ray encryption resources. You can use AWS Config to inventory X-Ray encryption resources, audit the X-Ray configuration history, and send notifications based on resource changes. For more information, see Tracking X-Ray encryption configuration changes with AWS Config.

Amazon CloudWatch Monitoring

You can use the X-Ray SDK for Java to publish unsampled Amazon CloudWatch metrics from your collected X-Ray segments. These metrics are derived from the segment's start and end time, and the error, fault and throttled status flags. Use these trace metrics to expose retries and dependency issues within subsegments. For more information, see AWS X-Ray metrics for the X-Ray SDK for Java.

Compliance validation for AWS X-Ray

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. 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 AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security Compliance & Governance – These solution implementation guides discuss architectural considerations and provide steps for deploying security and compliance features.
- HIPAA Eligible Services Reference – Lists HIPAA eligible services. Not all AWS services are HIPAA eligible.
- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Customer Compliance Guides – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).

- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see Security Hub controls reference.
- Amazon GuardDuty – This AWS service detects potential threats to your AWS accounts, workloads, containers, and data by monitoring your environment for suspicious and malicious activities. GuardDuty can help you address various compliance requirements, like PCI DSS, by meeting intrusion detection requirements mandated by certain compliance frameworks.
- AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Resilience in AWS X-Ray

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.

Infrastructure security in AWS X-Ray

As a managed service, AWS X-Ray 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 X-Ray 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.

Using AWS X-Ray with VPC endpoints

If you use Amazon Virtual Private Cloud (Amazon VPC) to host your AWS resources, you can establish a private connection between your VPC and X-Ray. This enables resources in your Amazon VPC to communicate with the X-Ray service without going through the public internet.

Amazon VPC is an AWS service that you can use to launch AWS resources in a virtual network that you define. With a VPC, you have control over your network settings, such as the IP address range, subnets, route tables, and network gateways. To connect your VPC to X-Ray, you define an interface VPC endpoint. The endpoint provides reliable, scalable connectivity to X-Ray without requiring an internet gateway, network address translation (NAT) instance, or VPN connection. For more information, see What Is Amazon VPC in the *Amazon VPC User Guide*.

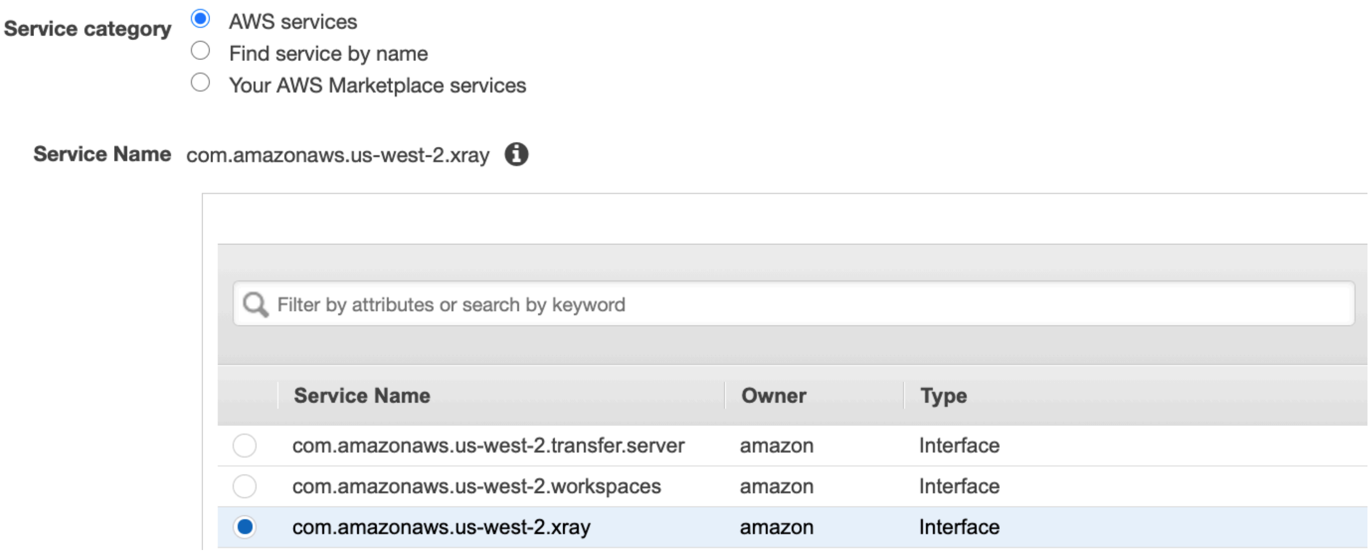
Interface VPC endpoints are powered by AWS PrivateLink, an AWS technology that enables private communication between AWS services by using an elastic network interface with private IP addresses. For more information, see the New – AWS PrivateLink for AWS services blog post and Getting Started in the *Amazon VPC User Guide*.

To ensure you can create a VPC endpoint for X-Ray in your chosen AWS Region, see Supported Regions.

Creating a VPC endpoint for X-Ray

To start using X-Ray with your VPC, create an interface VPC endpoint for X-Ray.

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. Navigate to **Endpoints** within the navigation pane and choose **Create Endpoint**.
3. Search for and select the name of the AWS X-Ray service: `com.amazonaws.`*`region`*`.xray`.

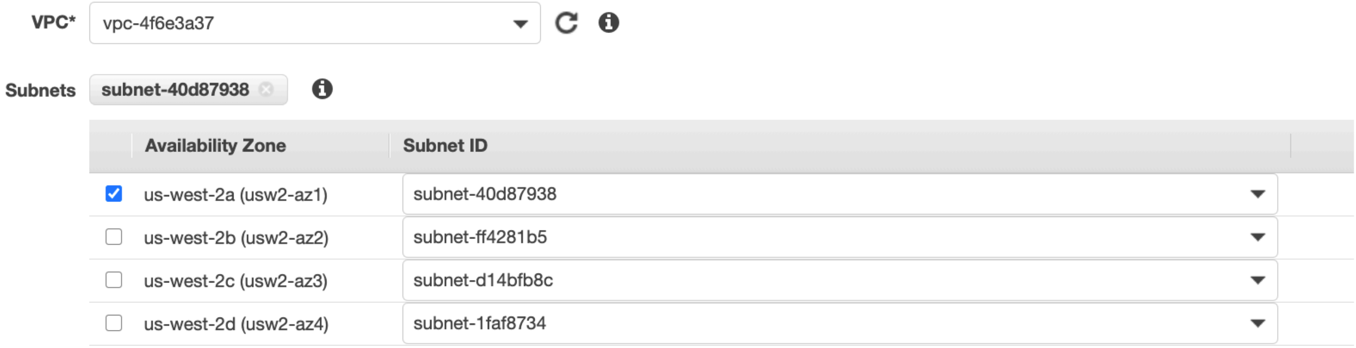


Service category
● AWS services
○ Find service by name
○ Your AWS Marketplace services

Service Name com.amazonaws.us-west-2.xray

	Service Name	Owner	Type
○	com.amazonaws.us-west-2.transfer.server	amazon	Interface
○	com.amazonaws.us-west-2.workspaces	amazon	Interface
●	com.amazonaws.us-west-2.xray	amazon	Interface

4. Select the VPC you want and then select a subnet in your VPC to use the interface endpoint. An endpoint network interface is created in the selected subnet. You can specify more than one subnet in different Availability Zones (as supported by the service) to help ensure that your interface endpoint is resilient to Availability Zone failures. If you do so, an interface network interface is created in each subnet that you specify.



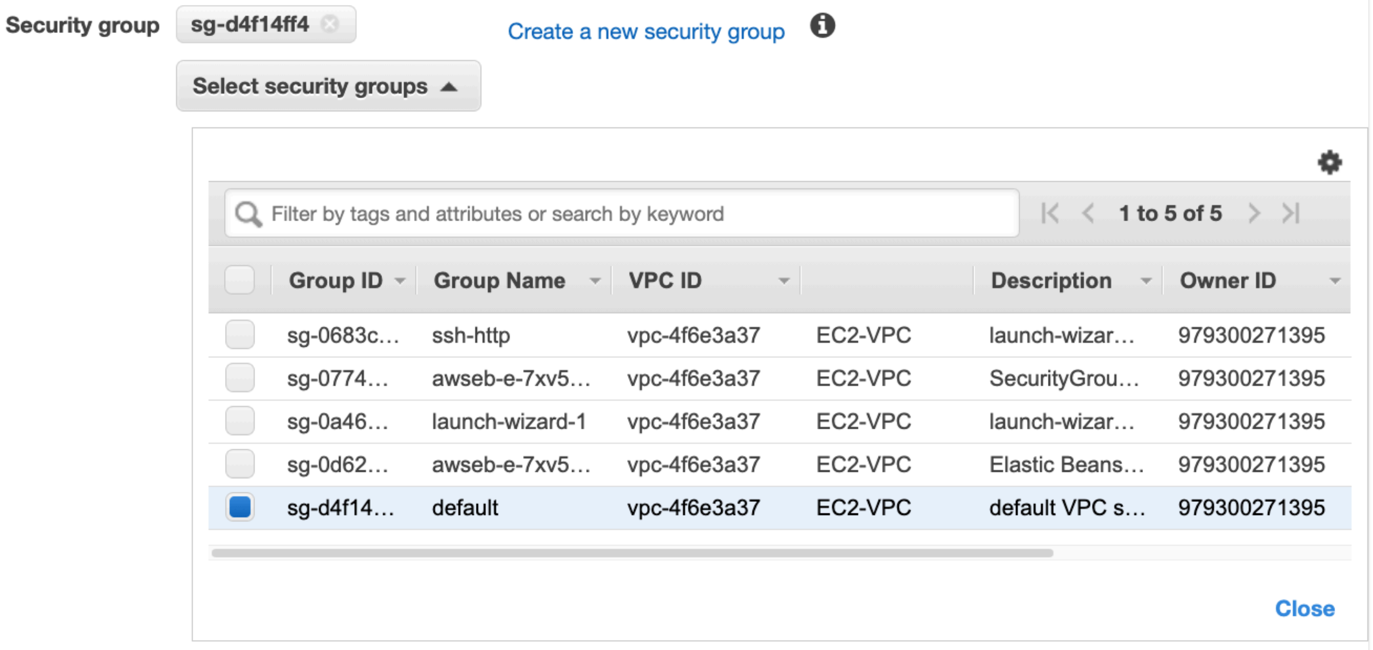
VPC* vpc-4f6e3a37

Subnets subnet-40d87938

	Availability Zone	Subnet ID
☑	us-west-2a (usw2-az1)	subnet-40d87938
□	us-west-2b (usw2-az2)	subnet-ff4281b5
□	us-west-2c (usw2-az3)	subnet-d14bfb8c
□	us-west-2d (usw2-az4)	subnet-1faf8734

5. (Optional) Private DNS is enabled by default for the endpoint, so that you can make requests to X-Ray using its default DNS hostname. You can choose to disable it.

6. Specify the security groups to associate with the endpoint network interface.


Create a new security group
Select security groups
Filter by tags and attributes or search by keyword
1 to 5 of 5
Close

Security group: sg-d4f14ff4

	Group ID	Group Name	VPC ID		Description	Owner ID
☐	sg-0683c…	ssh-http	vpc-4f6e3a37	EC2-VPC	launch-wizar…	979300271395
☐	sg-0774…	awseb-e-7xv5…	vpc-4f6e3a37	EC2-VPC	SecurityGrou…	979300271395
☐	sg-0a46…	launch-wizard-1	vpc-4f6e3a37	EC2-VPC	launch-wizar…	979300271395
☐	sg-0d62…	awseb-e-7xv5…	vpc-4f6e3a37	EC2-VPC	Elastic Beans…	979300271395
☑	sg-d4f14…	default	vpc-4f6e3a37	EC2-VPC	default VPC s…	979300271395

7. (Optional) Specify custom policy to control permissions to access the X-Ray service. By default, full access is allowed.

Controlling access to your X-Ray VPC endpoint

A VPC endpoint policy is an IAM resource policy that you attach to an endpoint when you create or modify the endpoint. If you don't attach a policy when you create an endpoint, Amazon VPC attaches a default policy for you that allows full access to the service. An endpoint policy doesn't override or replace IAM user policies or service-specific policies. It's a separate policy for controlling access from the endpoint to the specified service. Endpoint policies must be written in JSON format. For more information, see Controlling Access to Services with VPC Endpoints in the *Amazon VPC User Guide*.

VPC endpoint policy enables you to control permissions to various X-Ray actions. For example, you can create a policy to allow only PutTraceSegment and deny all other actions. This restricts workloads and services in the VPC to send only trace data to X-Ray and deny any other action such as retrieve data, change encryption config, or create/update groups.

The following is an example of an endpoint policy for X-Ray. This policy allows users connecting to X-Ray through the VPC to send segment data to X-Ray, and also prevents them from performing other X-Ray actions.

```
{"Statement": [
    {"Sid": "Allow PutTraceSegments",
      "Principal": "*",
      "Action": [
        "xray:PutTraceSegments"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

To edit the VPC endpoint policy for X-Ray

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Endpoints**.
3. If you haven't already created the endpoint for X-Ray, follow the steps in Creating a VPC endpoint for X-Ray.
4. Select the **com.amazonaws.*region*.xray** endpoint, and then choose the **Policy** tab.
5. Choose **Edit Policy**, and then make your changes.

Supported Regions

X-Ray currently supports VPC endpoints in the following AWS Regions:

- US East (Ohio)
- US East (N. Virginia)
- US West (N. California)
- US West (Oregon)
- Africa (Cape Town)
- Asia Pacific (Hong Kong)
- 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)
- AWS GovCloud (US-East)
- AWS GovCloud (US-West)

Cross-service confused deputy prevention

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way it should not otherwise have permission to access. To prevent this, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account.

We recommend using the `aws:SourceArn`, `aws:SourceAccount`, `aws:SourceOrgID`, and `aws:SourceOrgPaths` global condition context keys in resource policies to limit the permissions that xraylong gives another service to the resource. Use `aws:SourceArn` to associate only one resource with cross-service access. Use `aws:SourceAccount` to let any resource in that account be associated with the cross-service use. Use `aws:SourceOrgID` to allow any resource from any account within an organization be associated with the cross-service use. Use `aws:SourceOrgPaths` to associate any resource from accounts within an AWS Organizations path with the cross-service use. For more information about using and understanding paths, see Understand the AWS Organizations entity path.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full ARN of the resource. If you don't know the full ARN of the resource or if you are specifying multiple resources, use the `aws:SourceArn` global context condition key with wildcard characters (*) for the unknown portions of the ARN. For example, `arn:aws:servicename:*:123456789012:*`.

If the `aws:SourceArn` value does not contain the account ID, such as an Amazon S3 bucket ARN, you must use both `aws:SourceAccount` and `aws:SourceArn` to limit permissions.

To protect against the confused deputy problem at scale, use the `aws:SourceOrgID` or `aws:SourceOrgPaths` global condition context key with the organization ID or organization path of the resource in your resource-based policies. Policies that include the `aws:SourceOrgID` or `aws:SourceOrgPaths` key will automatically include the correct accounts and you don't have to manually update the policies when you add, remove, or move accounts in your organization.

The following example shows how you can use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in xray to prevent the confused deputy problem.

```
{
      "Sid": "BlockCrossAccountUnlessSameSource",
      "Effect": "Deny",
      "Principal": {
        "AWS": "*"
      },
      "Action": [
        "kms:Decrypt",
        "kms:GenerateDataKeyWithoutPlaintext"
      ],
      "Resource": "*",
      "Condition": {
        "StringNotEquals": {
          "aws:PrincipalAccount": "123456789012",
          "aws:SourceAccount": "123456789012"
        },
        "ArnNotLike": {
          "aws:SourceArn": "arn:*:*:*:123456789012:*"
        }
      }
    }
```

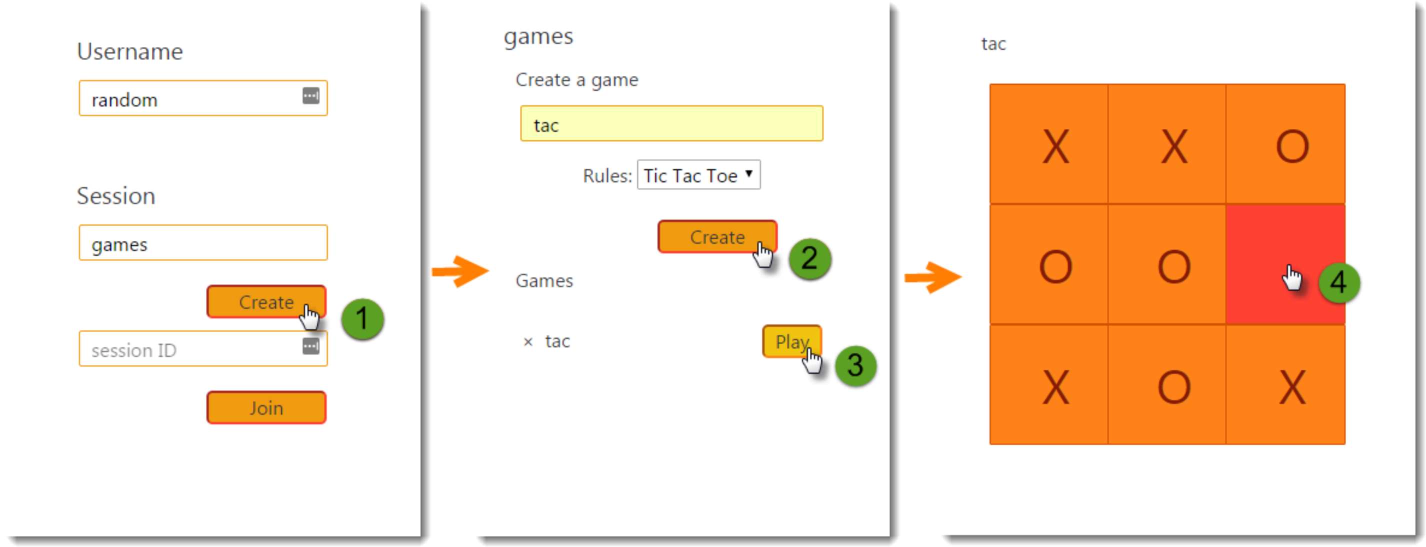
AWS X-Ray sample application

Note

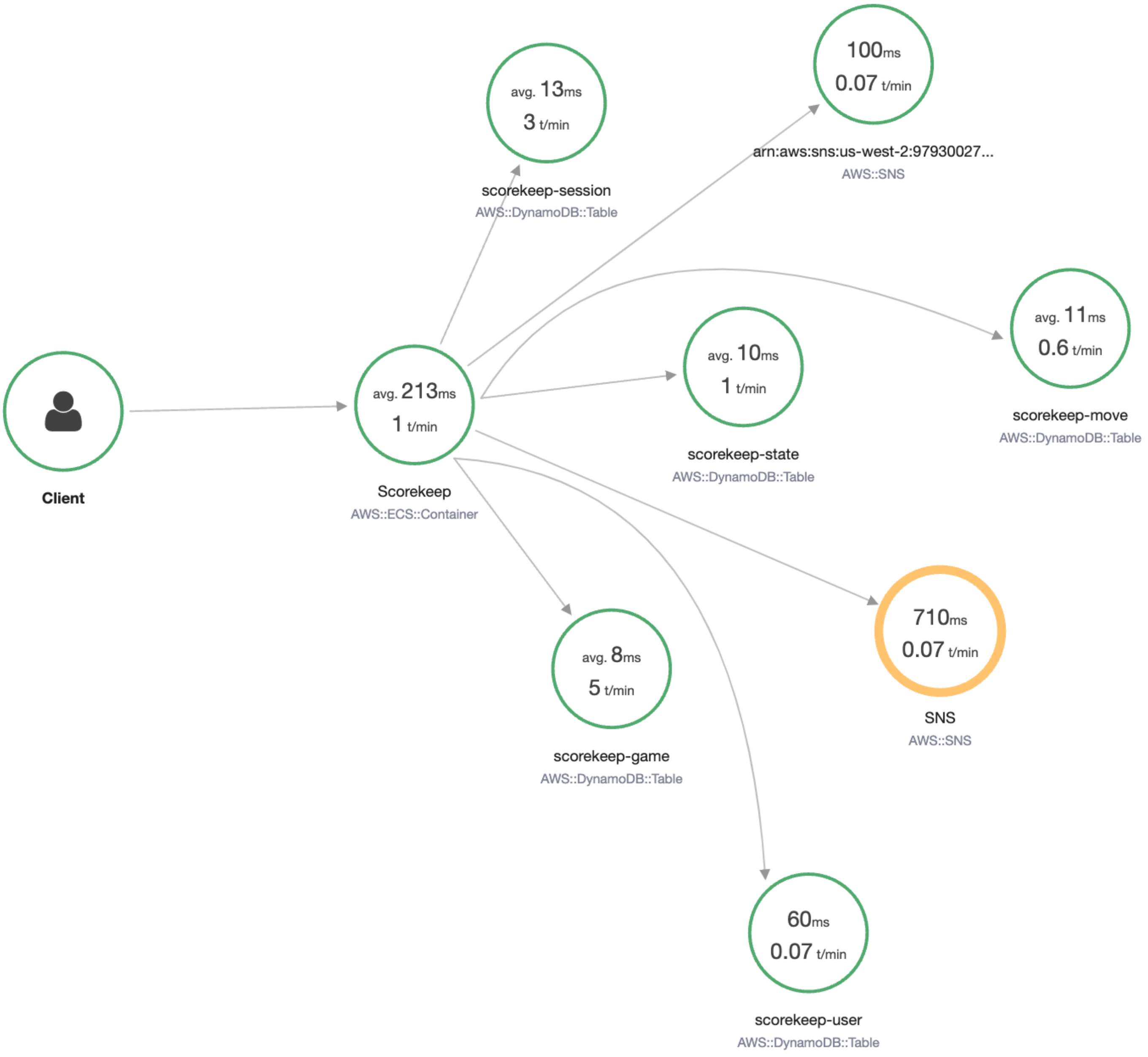
End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The AWS X-Ray eb-java-scorekeep sample app, available on GitHub, shows the use of the AWS X-Ray SDK to instrument incoming HTTP calls, DynamoDB SDK clients, and HTTP clients. The sample app uses AWS CloudFormation to create DynamoDB tables, compile Java code on instance, and run the X-Ray daemon without any additional configuration.

See the Scorekeep tutorial to start installing and using an instrumented sample application, using the AWS Management Console or the AWS CLI.



The sample includes a front-end web app, the API that it calls, and the DynamoDB tables that it uses to store data. Basic instrumentation with filters, plugins, and instrumented AWS SDK clients is shown in the project's `xray-gettingstarted` branch. This is the branch that you deploy in the getting started tutorial. Because this branch only includes the basics, you can diff it against the `master` branch to quickly understand the basics.



The sample application shows basic instrumentation in these files:

- **HTTP request filter –** `WebConfig.java`
- **AWS SDK client instrumentation –** `build.gradle`

The `xray` branch of the application includes the use of HTTPClient, Annotations, SQL queries, custom subsegments, an instrumented AWS Lambda function, and instrumented initialization code and scripts.

To support user log-in and AWS SDK for JavaScript use in the browser, the `xray-cognito` branch adds Amazon Cognito to support user authentication and authorization. With credentials retrieved from Amazon Cognito, the web app also sends trace data to X-Ray to record request information from the client's point of view. The browser client appears as its own node on the trace map, and records additional information, including the URL of the page that the user is viewing, and the user's ID.

Finally, the `xray-worker` branch adds an instrumented Python Lambda function that runs independently, processing items from an Amazon SQS queue. Scorekeep adds an item to the queue each time a game ends. The Lambda worker, triggered by CloudWatch Events, pulls items from the queue every few minutes and processes them to store game records in Amazon S3 for analysis.

Topics

- Getting started with the Scorekeep sample application
- Manually instrumenting AWS SDK clients
- Creating additional subsegments
- Recording annotations, metadata, and user IDs
- Instrumenting outgoing HTTP calls
- Instrumenting calls to a PostgreSQL database
- Instrumenting AWS Lambda functions
- Instrumenting startup code
- Instrumenting scripts
- Instrumenting a web app client
- Using instrumented clients in worker threads

Getting started with the Scorekeep sample application

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive

updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

This tutorial uses the `xray-gettingstarted` branch of the Scorekeep sample application, which uses AWS CloudFormation to create and configure the resources that run the sample application and X-Ray daemon on Amazon ECS. The application uses the Spring framework to implement a JSON web API and the AWS SDK for Java to persist data to Amazon DynamoDB. A servlet filter in the application instruments all incoming requests served by the application, and a request handler on the AWS SDK client instruments downstream calls to DynamoDB.

You can follow this tutorial using either the AWS Management Console or the AWS CLI.

Sections

- Prerequisites
- Install the Scorekeep application using CloudFormation
- Generate trace data
- View the trace map in the AWS Management Console
- Configuring Amazon SNS notifications
- Explore the sample application
- Optional: Least privilege policy
- Clean up
- Next steps

Prerequisites

This tutorial uses AWS CloudFormation to create and configure the resources that run the sample application and X-Ray daemon. The following prerequisites are required to install and run through the tutorial:

1. If you use an IAM user with limited permissions, add the following user policies in the IAM console:

   - `AWSCloudFormationFullAccess` – to access and use CloudFormation

- `AmazonS3FullAccess` – to upload a template file to CloudFormation using the AWS Management Console
- `IAMFullAccess` – to create the Amazon ECS and Amazon EC2 instance roles
- `AmazonEC2FullAccess` – to create the Amazon EC2 resources
- `AmazonDynamoDBFullAccess` – to create the DynamoDB tables
- `AmazonECS_FullAccess` – to create Amazon ECS resources
- `AmazonSNSFullAccess` – to create the Amazon SNS topic
- `AWSXrayReadOnlyAccess` – for permission to view the trace map and traces in the X-Ray console

2. To run through the tutorial using the AWS CLI, install the CLI version 2.7.9 or later, and configure the CLI with the user from the previous step. Make sure the region is configured when configuring the AWS CLI with the user. If a region is not configured, you will need to append `--region` *`AWS-REGION`* to every CLI command.
3. Ensure that Git is installed, in order to clone the sample application repo.
4. Use the following code example to clone the `xray-gettingstarted` branch of the Scorekeep repository:

```
git clone https://github.com/aws-samples/eb-java-scorekeep.git xray-scorekeep -b
 xray-gettingstarted
```

Install the Scorekeep application using CloudFormation

AWS Management Console

Install the sample application using the AWS Management Console

1. Open the CloudFormation console
2. Choose **Create stack** and then choose **With new resources** from the drop-down menu.
3. In the **Specify template** section, choose **Upload a template file**.
4. Select **Choose file**, navigate to the `xray-scorekeep/cloudformation` folder that was created when you cloned the git repo, and choose the `cf-resources.yaml` file.
5. Choose **Next** to continue.

6. Enter `scorekeep` into the **Stack name** textbox, and then choose **Next** at the bottom of the page to continue. Note that the rest of this tutorial assumes the stack is named `scorekeep`.
7. Scroll to the bottom of the **Configure stack options** page and choose **Next** to continue.
8. Scroll to the bottom of the **Review** page, choose the check-box acknowledging that CloudFormation may create IAM resources with custom names, and choose **Create stack**.
9. The CloudFormation stack is now being created. The stack status will be `CREATE_IN_PROGRESS` for about five minutes before changing to `CREATE_COMPLETE`. The status will refresh periodically, or you can refresh the page.

AWS CLI

Install the sample application using the AWS CLI

1. Navigate to the `cloudformation` folder of the `xray-scorekeep` repository that you cloned earlier in the tutorial:

   ```
   cd xray-scorekeep/cloudformation/
   ```

2. Enter the following AWS CLI command to create the CloudFormation stack:

   ```
   aws cloudformation create-stack --stack-name scorekeep --capabilities
    "CAPABILITY_NAMED_IAM" --template-body file://cf-resources.yaml
   ```

3. Wait until the CloudFormation stack status is `CREATE_COMPLETE`, which will take about five minutes. Use the following AWS CLI command to check on the status:

   ```
   aws cloudformation describe-stacks --stack-name scorekeep --query
    "Stacks[0].StackStatus"
   ```

Generate trace data

The sample application includes a front-end web app. Use the web app to generate traffic to the API and send trace data to X-Ray. First, retrieve the web app URL using the AWS Management Console or the AWS CLI:

AWS Management Console

Find the application URL using the AWS Management Console

1. Open the CloudFormation console
2. Choose the `scorekeep` stack from the list.
3. Choose the **Outputs** tab on the `scorekeep` stack page, and choose the `LoadBalancerUrl` URL link to open the web application.

AWS CLI

Find the application URL using the AWS CLI

1. Use the following command to display the URL of the web application:

```
aws cloudformation describe-stacks --stack-name scorekeep --query
 "Stacks[0].Outputs[0].OutputValue"
```

2. Copy this URL and open in a browser to display the Scorekeep web application.

Use the web application to generate trace data

1. Choose **Create** to create a user and session.
2. Type a **game name**, set the **Rules** to **Tic Tac Toe**, and then choose **Create** to create a game.
3. Choose **Play** to start the game.
4. Choose a tile to make a move and change the game state.

Each of these steps generates HTTP requests to the API, and downstream calls to DynamoDB to read and write user, session, game, move, and state data.

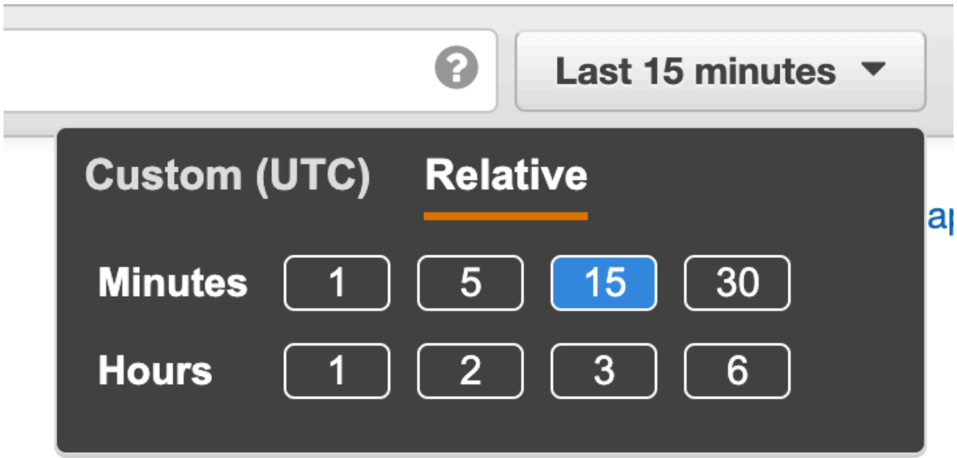
View the trace map in the AWS Management Console

You can see the trace map and traces generated by the sample application in the X-Ray and CloudWatch consoles.

X-Ray console

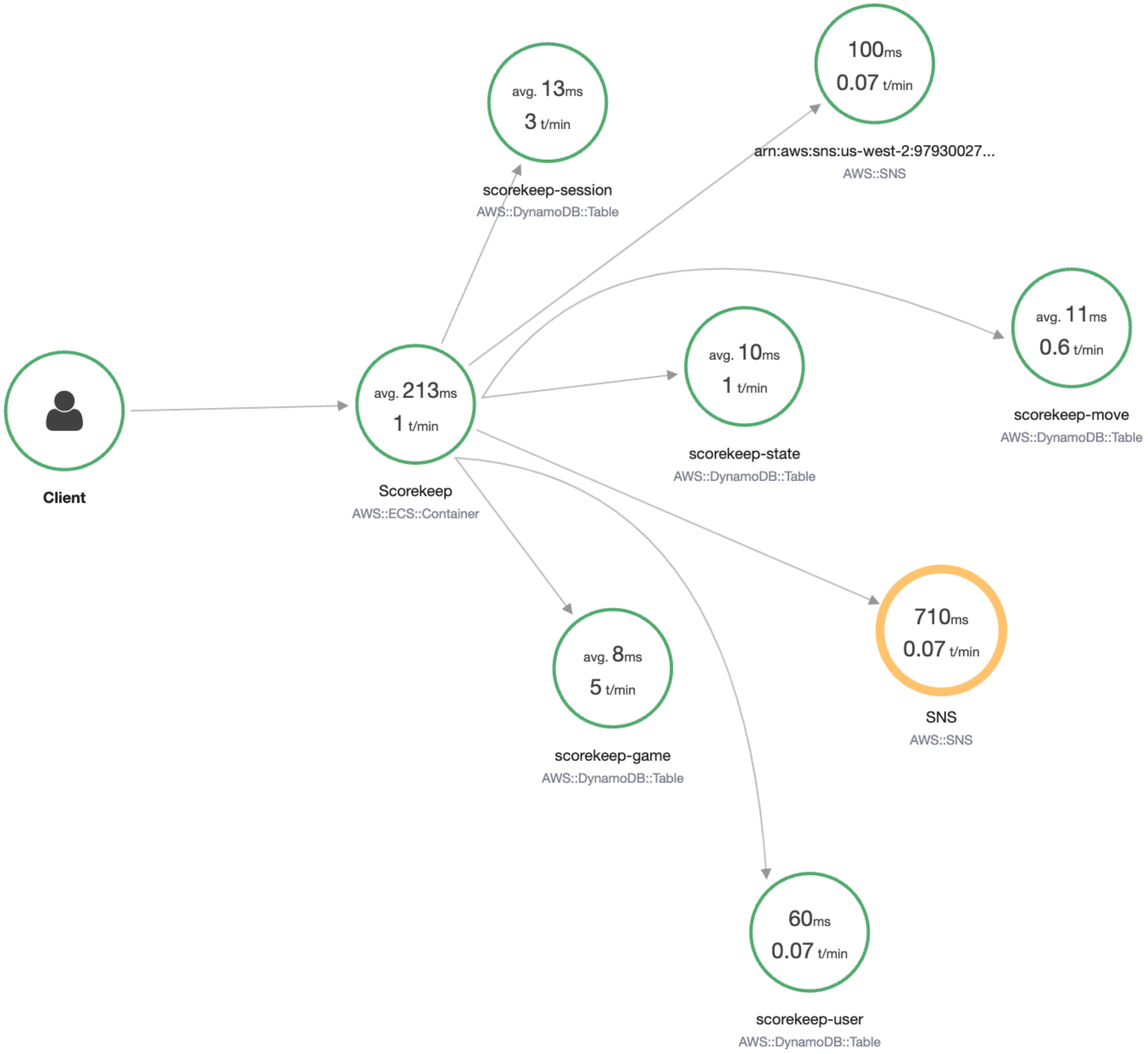
Use the X-Ray console

1. Open the trace map page of the X-Ray console.
2. The console shows a representation of the service graph that X-Ray generates from the trace data sent by the application. Be sure to adjust the time period of the trace map if needed, to make sure that it will display all traces since you first started the web application.



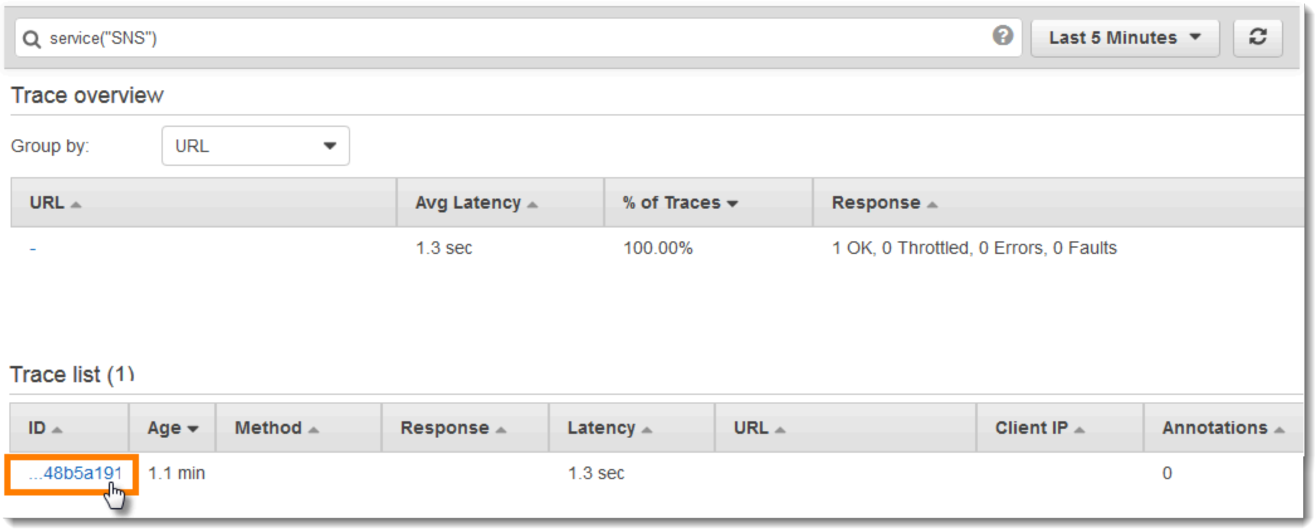
The trace map shows the web app client, the API running in Amazon ECS, and each DynamoDB table that the application uses. Every request to the application, up to a configurable maximum number of requests per second, is traced as it hits the API, generates requests to downstream services, and completes.

You can choose any node in the service graph to view traces for requests that generated traffic to that node. Currently, the Amazon SNS node is yellow. Drill down to find out why.



To find the cause of the error

1. Choose the node named **SNS**. The node details panel is displayed.
2. Choose **View traces** to access the **Trace overview** screen.
3. Choose the trace from the **Trace list**. This trace doesn't have a method or URL because it was recorded during startup instead of in response to an incoming request.



service("SNS")

Trace overview

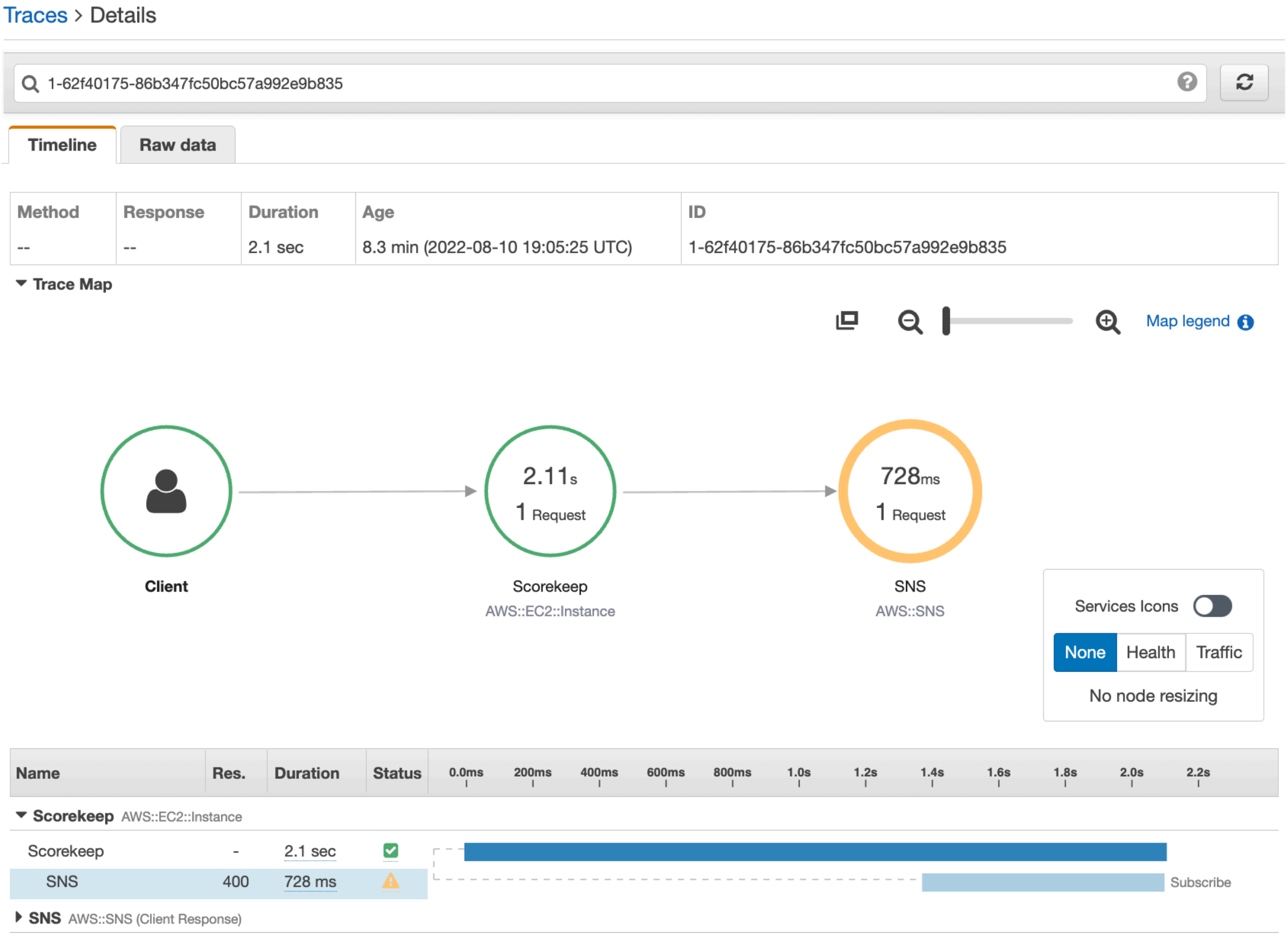
Group by: URL

URL	Avg Latency	% of Traces	Response
-	1.3 sec	100.00%	1 OK, 0 Throttled, 0 Errors, 0 Faults

Trace list (1)

ID	Age	Method	Response	Latency	URL	Client IP	Annotations
...48b5a191	1.1 min			1.3 sec			0

4. Choose the error status icon within the Amazon SNS segment at the bottom of the page, to open the **Exceptions** page for the SNS subsegment.



Traces > Details

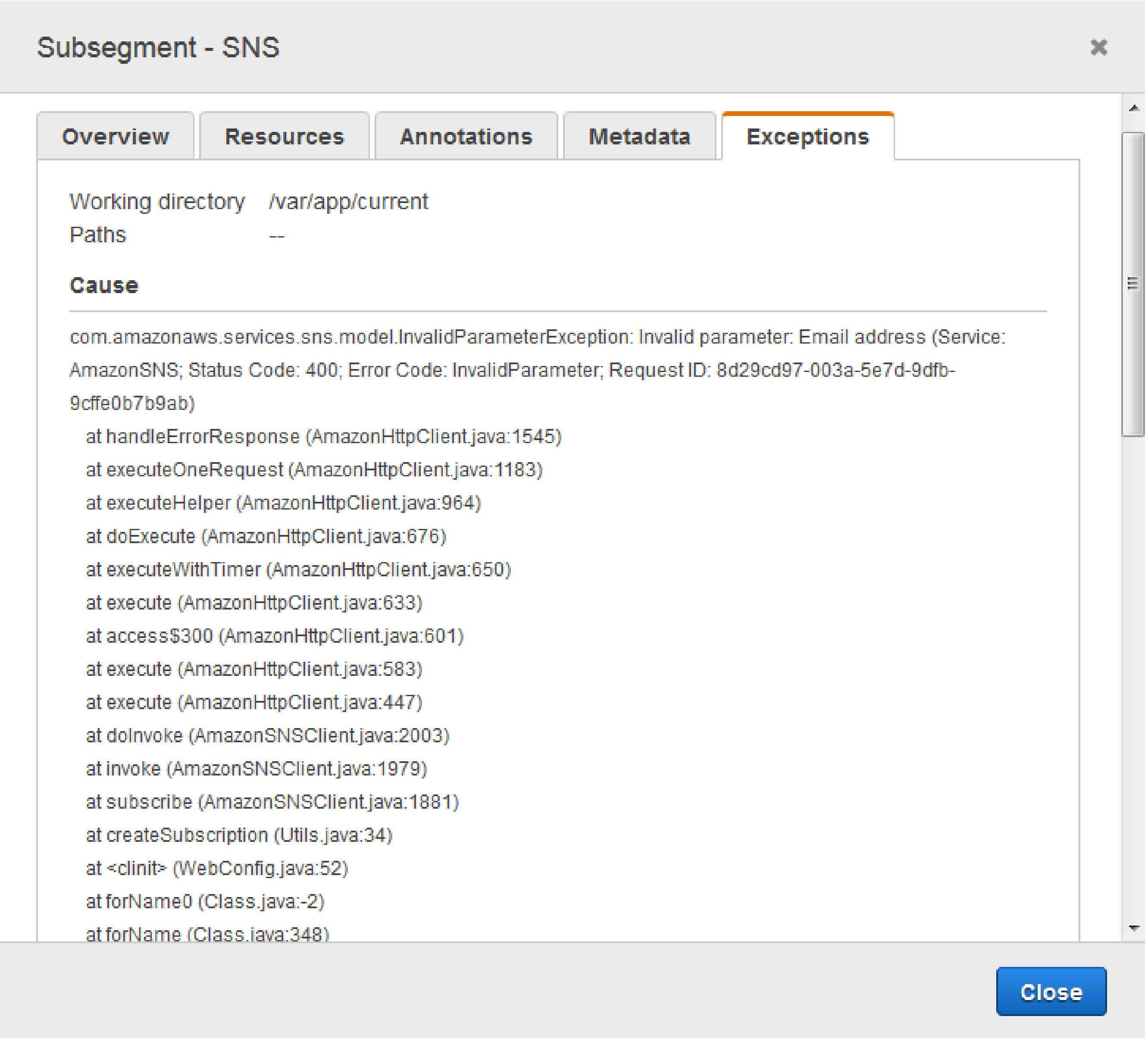
1-62f40175-86b347fc50bc57a992e9b835

Method	Response	Duration	Age	ID
--	--	2.1 sec	8.3 min (2022-08-10 19:05:25 UTC)	1-62f40175-86b347fc50bc57a992e9b835

Trace Map

Name	Res.	Duration	Status
Scorekeep AWS::EC2::Instance			
Scorekeep	-	2.1 sec	✓
SNS	400	728 ms	⚠
SNS AWS::SNS (Client Response)			

5. The X-Ray SDK automatically captures exceptions thrown by instrumented AWS SDK clients and records the stack trace.



Working directory /var/app/current

Paths --

Cause

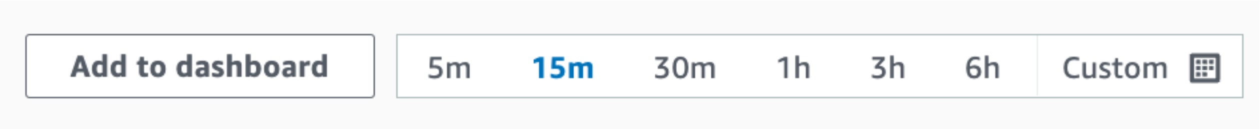
```
com.amazonaws.services.sns.model.InvalidParameterException: Invalid parameter: Email address (Service:
AmazonSNS; Status Code: 400; Error Code: InvalidParameter; Request ID: 8d29cd97-003a-5e7d-9dfb-
9cffe0b7b9ab)
  at handleErrorResponse (AmazonHttpClient.java:1545)
  at executeOneRequest (AmazonHttpClient.java:1183)
  at executeHelper (AmazonHttpClient.java:964)
  at doExecute (AmazonHttpClient.java:676)
  at executeWithTimer (AmazonHttpClient.java:650)
  at execute (AmazonHttpClient.java:633)
  at access$300 (AmazonHttpClient.java:601)
  at execute (AmazonHttpClient.java:583)
  at execute (AmazonHttpClient.java:447)
  at doInvoke (AmazonSNSClient.java:2003)
  at invoke (AmazonSNSClient.java:1979)
  at subscribe (AmazonSNSClient.java:1881)
  at createSubscription (Utils.java:34)
  at <clinit> (WebConfig.java:52)
  at forName0 (Class.java:-2)
  at forName (Class.java:348)
```

CloudWatch console

Use the CloudWatch console

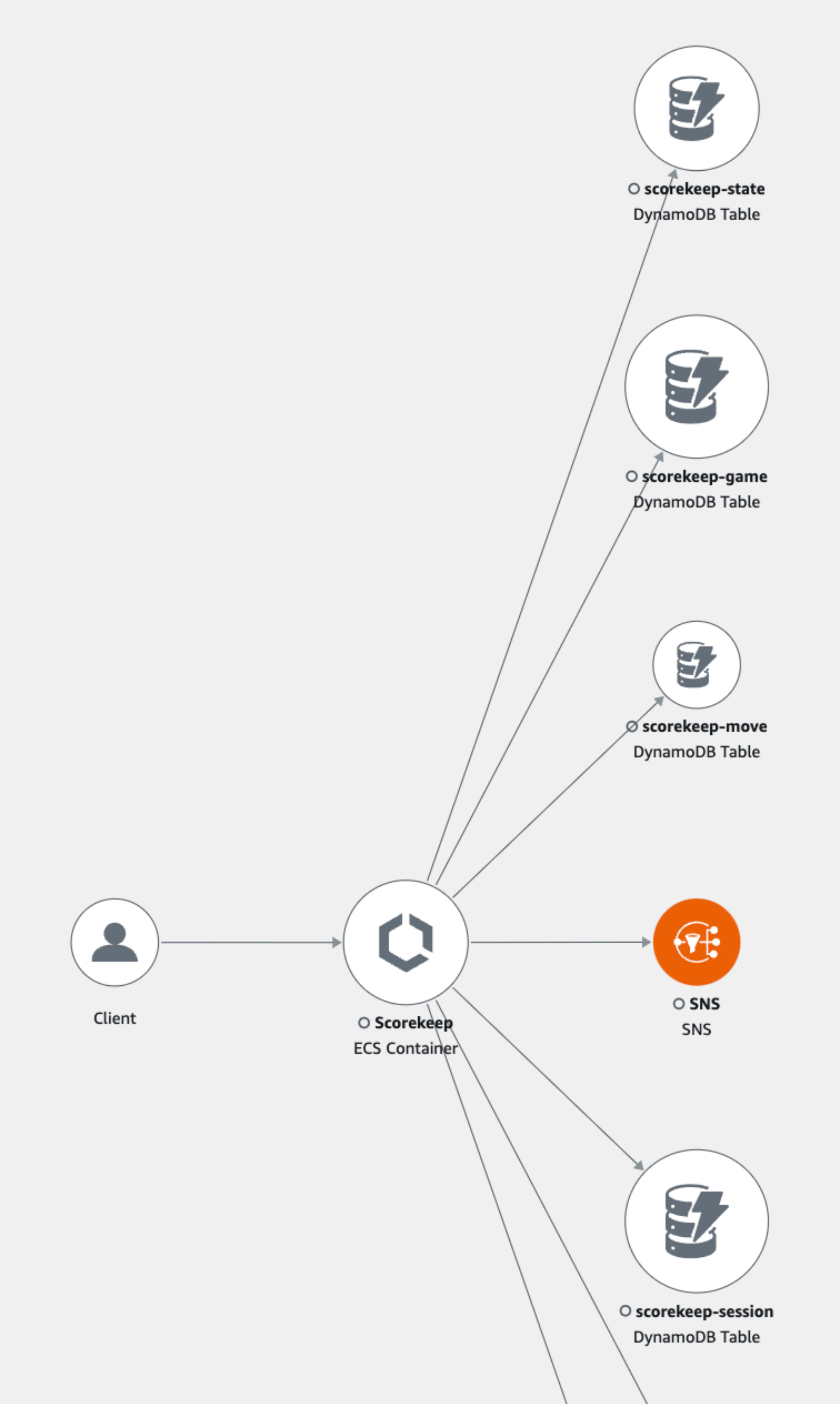
1. Open the X-Ray trace map page of the CloudWatch console.
2. The console shows a representation of the service graph that X-Ray generates from the trace data sent by the application. Be sure to adjust the time period of the trace

map if needed, to make sure that it will display all traces since you first started the web application.



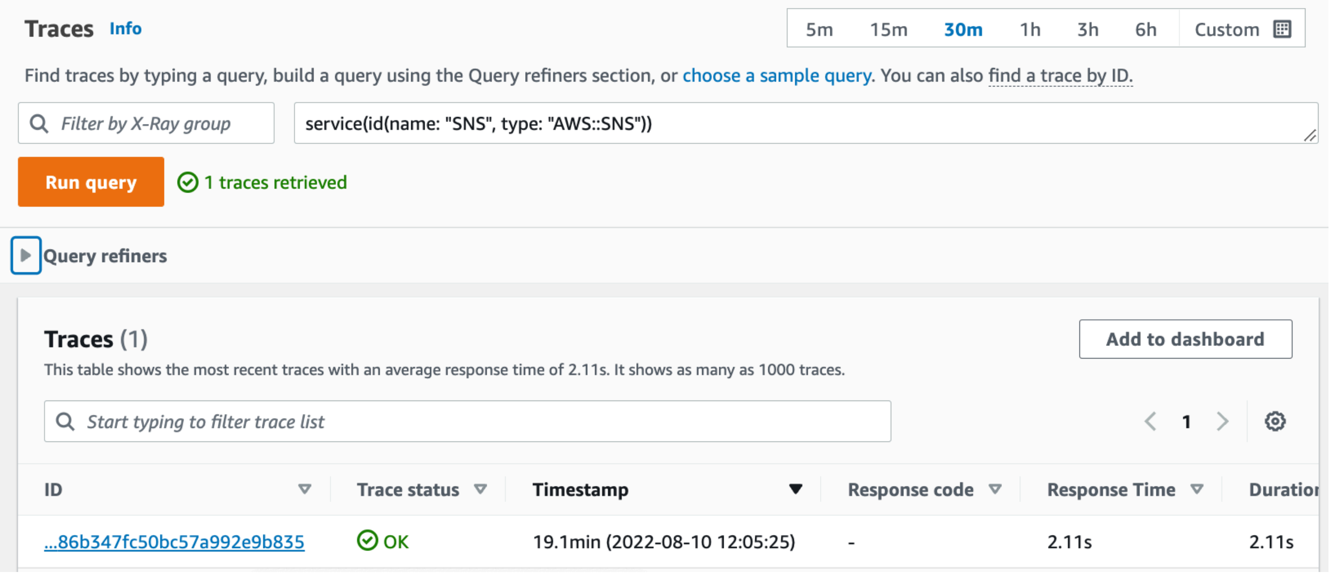
The trace map shows the web app client, the API running in Amazon EC2, and each DynamoDB table that the application uses. Every request to the application, up to a configurable maximum number of requests per second, is traced as it hits the API, generates requests to downstream services, and completes.

You can choose any node in the service graph to view traces for requests that generated traffic to that node. Currently, the Amazon SNS node is orange. Drill down to find out why.


scorekeep-state
DynamoDB Table
scorekeep-game
DynamoDB Table
scorekeep-move
DynamoDB Table
Client
Scorekeep
ECS Container
SNS
SNS
scorekeep-session
DynamoDB Table

To find the cause of the error

1. Choose the node named **SNS**. The SNS node details panel is displayed below the map.
2. Choose **View traces** to access the **Traces** page.
3. Add the bottom of the page, choose the trace from the **Traces** list. This trace doesn't have a method or URL because it was recorded during startup instead of in response to an incoming request.



Find traces by typing a query, build a query using the Query refiners section, or choose a sample query. You can also find a trace by ID.

service(id(name: "SNS", type: "AWS::SNS"))

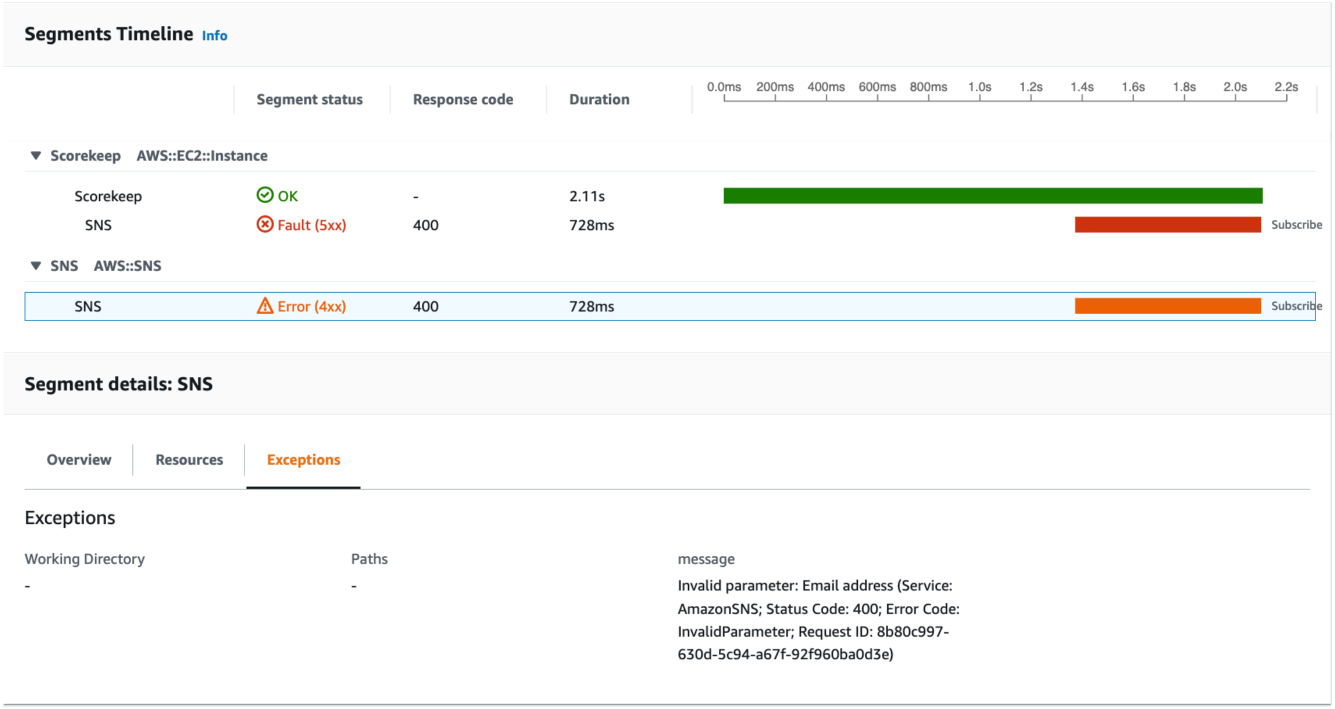
1 traces retrieved

Traces (1)

This table shows the most recent traces with an average response time of 2.11s. It shows as many as 1000 traces.

ID	Trace status	Timestamp	Response code	Response Time	Duratio
...86b347fc50bc57a992e9b835	OK	19.1min (2022-08-10 12:05:25)	-	2.11s	2.11s

4. Choose the Amazon SNS subsegment at the bottom of the segments timeline, and choose the **Exceptions** tab for the SNS subsegment to view the exception details.



	Segment status	Response code	Duration
▼ Scorekeep AWS::EC2::Instance			
Scorekeep	OK	-	2.11s
SNS	Fault (5xx)	400	728ms
▼ SNS AWS::SNS			
SNS	Error (4xx)	400	728ms

Segment details: SNS

Exceptions

Working Directory	Paths	message
-	-	Invalid parameter: Email address (Service: AmazonSNS; Status Code: 400; Error Code: InvalidParameter; Request ID: 8b80c997-630d-5c94-a67f-92f960ba0d3e)

The cause indicates that the email address provided in a call to `createSubscription` made in the `WebConfig` class was invalid. In the next section, we'll fix that.

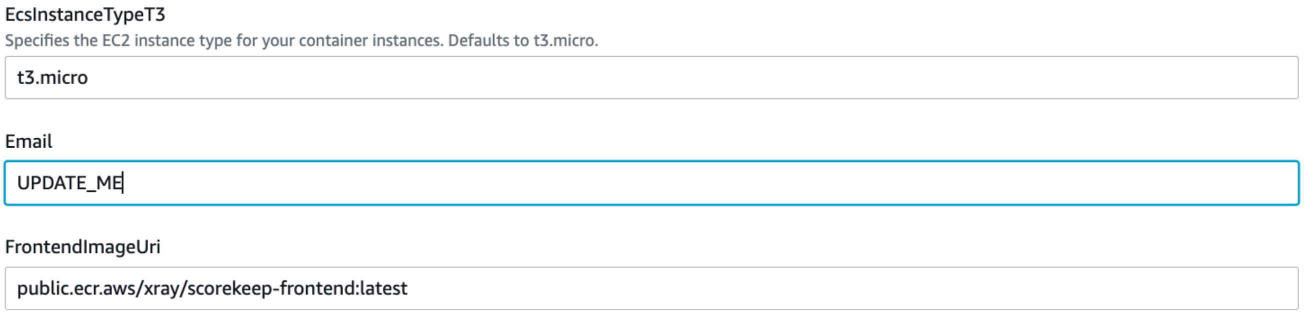
Configuring Amazon SNS notifications

Scorekeep uses Amazon SNS to send notifications when users complete a game. When the application starts up, it tries to create a subscription for an email address defined in a CloudFormation stack parameter. That call is currently failing. Configure a notification email to enable notifications, and resolve the failures highlighted in the trace map.

AWS Management Console

To configure Amazon SNS notifications using the AWS Management Console

1. Open the [CloudFormation console](#)
2. Choose the radio button next to the `scorekeep` stack name in the list, and then choose **Update**.
3. Make sure that **Use current template** is chosen, and then click **Next** on the **Update stack** page.
4. Find the **Email** parameter in the list, and replace the default value with a valid email address.



EcsInstanceTypeT3
Specifies the EC2 instance type for your container instances. Defaults to t3.micro.
t3.micro

Email
UPDATE_ME

FrontendImageUri
public.ecr.aws/xray/scorekeep-frontend:latest

5. Scroll to the bottom of the page and choose **Next**.
6. Scroll to the bottom of the **Review** page, choose the check-box acknowledging that CloudFormation may create IAM resources with custom names, and choose **Update stack**.
7. The CloudFormation stack is now being updated. The stack status will be `UPDATE_IN_PROGRESS` for about five minutes before changing to `UPDATE_COMPLETE`. The status will refresh periodically, or you can refresh the page.

AWS CLI

To configure Amazon SNS notifications using the AWS CLI

1. Navigate to the `xray-scorekeep/cloudformation/` folder you previously created, and open the `cf-resources.yaml` file in a text editor.
2. Find the `Default` value within the **Email** parameter and change it from *`UPDATE_ME`* to a valid email address.

   ```
   Parameters:
     Email:
       Type: String
       Default: UPDATE_ME # <- change to a valid abc@def.xyz email address
   ```

3. From the `cloudformation` folder, update the CloudFormation stack with the following AWS CLI command:

   ```
   aws cloudformation update-stack --stack-name scorekeep --capabilities
    "CAPABILITY_NAMED_IAM" --template-body file://cf-resources.yaml
   ```

4. Wait until the CloudFormation stack status is `UPDATE_COMPLETE`, which will take a few minutes. Use the following AWS CLI command to check on the status:

   ```
   aws cloudformation describe-stacks --stack-name scorekeep --query
    "Stacks[0].StackStatus"
   ```

When the update completes, Scorekeep restarts and creates a subscription to the SNS topic. Check your email and confirm the subscription to see updates when you complete a game. Open the trace map to verify that the calls to SNS are no longer failing.

Explore the sample application

The sample application is an HTTP web API in Java that is configured to use the X-Ray SDK for Java. When you deploy the application with the CloudFormation template, it creates the DynamoDB tables, Amazon ECS Cluster, and other services required to run Scorekeep on ECS. A task definition file for ECS is created through CloudFormation. This file defines the container images used per task in an ECS cluster. These images are obtained from the official X-Ray public ECR. The scorekeep API container image has the API compiled with Gradle. The container image of the Scorekeep frontend container serves the frontend using the nginx proxy server. This server routes requests to paths starting with /api to the API.

To instrument incoming HTTP requests, the application adds the `TracingFilter` provided by the SDK.

Example src/main/java/scorekeep/WebConfig.java - servlet filter

```
import javax.servlet.Filter;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;
...

@Configuration
public class WebConfig {

  @Bean
  public Filter TracingFilter() {
    return new AWSXRayServletFilter("Scorekeep");
  }
...
```

This filter sends trace data about all incoming requests that the application serves, including request URL, method, response status, start time, and end time.

The application also makes downstream calls to DynamoDB using the AWS SDK for Java. To instrument these calls, the application simply takes the AWS SDK-related submodules as dependencies, and the X-Ray SDK for Java automatically instruments all AWS SDK clients.

The application uses `Docker` to build the source code on-instance with the `Gradle Docker Image` and the `Scorekeep API Dockerfile` file to run the executable JAR that Gradle generates at its ENTRYPOINT.

Example use of Docker to build via Gradle Docker Image

```
docker run --rm -v /PATH/TO/SCOREKEEP_REPO/home/gradle/project -w /home/gradle/project
 gradle:4.3 gradle build
```

Example Dockerfile ENTRYPOINT

```
ENTRYPOINT [ "sh", "-c", "java -Dserver.port=5000 -jar scorekeep-api-1.0.0.jar" ]
```

The `build.gradle` file downloads the SDK submodules from Maven during compilation by declaring them as dependencies.

Example build.gradle -- dependencies

```
...
dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile('org.springframework.boot:spring-boot-starter-test')
    compile('com.amazonaws:aws-java-sdk-dynamodb')
    compile("com.amazonaws:aws-xray-recorder-sdk-core")
    compile("com.amazonaws:aws-xray-recorder-sdk-aws-sdk")
    compile("com.amazonaws:aws-xray-recorder-sdk-aws-sdk-instrumentor")
    ...
}
dependencyManagement {
    imports {
        mavenBom("com.amazonaws:aws-java-sdk-bom:1.11.67")
        mavenBom("com.amazonaws:aws-xray-recorder-sdk-bom:2.11.0")
    }
}
```

The core, AWS SDK, and AWS SDK Instrumentor submodules are all that's required to automatically instrument any downstream calls made with the AWS SDK.

To relay the raw segment data to the X-Ray API, the X-Ray daemon is required to listen for traffic on UDP port 2000. To do so, the application has the X-Ray daemon run in a container that is deployed alongside the Scorekeep application on ECS as a *sidecar container*. Check out the X-Ray daemon topic for more information.

Example X-Ray Daemon Container Definition in an ECS Task Definition

```
...
Resources:
  ScorekeepTaskDefinition:
    Type: AWS::ECS::TaskDefinition
    Properties:
      ContainerDefinitions:
      ...

      - Cpu: '256'
        Essential: true
        Image: amazon/aws-xray-daemon
        MemoryReservation: '128'
        Name: xray-daemon
        PortMappings:
          - ContainerPort: '2000'
            HostPort: '2000'
            Protocol: udp
      ...
```

The X-Ray SDK for Java provides a class named `AWSXRay` that provides the global recorder, a `TracingHandler` that you can use to instrument your code. You can configure the global recorder to customize the `AWSXRayServletFilter` that creates segments for incoming HTTP calls. The sample includes a static block in the `WebConfig` class that configures the global recorder with plugins and sampling rules.

Example src/main/java/scorekeep/WebConfig.java - recorder

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;
import com.amazonaws.xray.plugins.ECSPlugin;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;
...

@Configuration
public class WebConfig {
  ...

  static {
    AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder.standard().withPlugin(new
 ECSPlugin()).withPlugin(new EC2Plugin());

    URL ruleFile = WebConfig.class.getResource("/sampling-rules.json");
    builder.withSamplingStrategy(new LocalizedSamplingStrategy(ruleFile));

    AWSXRay.setGlobalRecorder(builder.build());
    ...

  }
}
```

This example uses the builder to load sampling rules from a file named `sampling-rules.json`. Sampling rules determine the rate at which the SDK records segments for incoming requests.

Example src/main/java/resources/sampling-rules.json

```
{
  "version": 1,
  "rules": [
    {
      "description": "Resource creation.",
      "service_name": "*",
      "http_method": "POST",
      "url_path": "/api/*",
      "fixed_target": 1,
      "rate": 1.0
    },
    {
      "description": "Session polling.",
      "service_name": "*",
      "http_method": "GET",
      "url_path": "/api/session/*",
      "fixed_target": 0,
      "rate": 0.05
    },
    {
      "description": "Game polling.",
      "service_name": "*",
      "http_method": "GET",
      "url_path": "/api/game/*/*",
      "fixed_target": 0,
      "rate": 0.05
    },
    {
      "description": "State polling.",
      "service_name": "*",
      "http_method": "GET",
      "url_path": "/api/state/*/*/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

The sampling rules file defines four custom sampling rules and the default rule. For each incoming request, the SDK evaluates the custom rules in the order in which they are defined. The SDK applies the first rule that matches the request's method, path, and service name. For Scorekeep, the first rule catches all POST requests (resource creation calls) by applying a fixed target of one request per second and a rate of 1.0, or 100 percent of requests after the fixed target is satisfied.

The other three custom rules apply a five percent rate with no fixed target to session, game, and state reads (GET requests). This minimizes the number of traces for periodic calls that the front end makes automatically every few seconds to ensure the content is up to date. For all other requests, the file defines a default rate of one request per second and a rate of 10 percent.

The sample application also shows how to use advanced features such as manual SDK client instrumentation, creating additional subsegments, and outgoing HTTP calls. For more information, see AWS X-Ray sample application.

Optional: Least privilege policy

The Scorekeep ECS containers access resources using full access policies, such as `AmazonSNSFullAccess` and `AmazonDynamoDBFullAccess`. Using full access policies is not the best practice for production applications. The following example updates the DynamoDB IAM policy to improve the security of the application. To learn more about security best practices in IAM policies, see Identity and access management for AWS X-Ray.

Example cf-resources.yaml template ECSTaskRole definition

```
ECSTaskRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          -
            Effect: "Allow"
            Principal:
              Service:
                - "ecs-tasks.amazonaws.com"
            Action:
              - "sts:AssumeRole"
      ManagedPolicyArns:
        - "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess"
        - "arn:aws:iam::aws:policy/AmazonSNSFullAccess"
        - "arn:aws:iam::aws:policy/AWSXrayFullAccess"
      RoleName: "scorekeepRole"
```

To update your policy, first you identify the ARN of your DynamoDB resources. Then you use the ARN in a custom IAM policy. Finally, you apply that policy to your instance profile.

To identify the ARN of your DynamoDB resource:

1. Open the DynamoDB console.
2. Choose **Tables** from the left navigation bar.
3. Choose any of the `scorekeep-*` to display the table detail page.
4. Under the **Overview** tab, choose **Additional info** to expand the section and view the Amazon Resource Name (ARN). Copy this value.
5. Insert the ARN into the following IAM policy, replacing the `AWS_REGION` and `AWS_ACCOUNT_ID` values with your specific region and account ID. This new policy allows only the actions specified, instead of the `AmazonDynamoDBFullAccess` policy which allows any action.

   Example

   The tables that the application creates follow a consistent naming convention. You can use the `scorekeep-*` format to indicate all Scorekeep tables.

Change your IAM policy

1. Open the Scorekeep task role (scorekeepRole) from the IAM console.
2. Choose the check box next to the `AmazonDynamoDBFullAccess` policy and choose **Remove** to remove this policy.
3. Choose **Add permissions**, and then **Attach policies**, and finally **Create policy**.
4. Choose the **JSON** tab and paste in the policy created above.
5. Choose **Next: Tags** at the bottom of the page.
6. Choose **Next: Review** at the bottom of the page.
7. For **Name**, assign a name for the policy.

8. Choose **Create policy** at the bottom of the page.
9. Attach the newly created policy to the `scorekeepRole` role. It may take a few minutes for the attached policy to take effect.

If you have attached the new policy to the `scorekeepRole` role, you must detach it before deleting the CloudFormation stack, since this attached policy will block the stack from being deleted. The policy can be automatically detached by deleting the policy.

Remove your custom IAM policy

1. Open the IAM console.
2. Choose **Policies** from the left navigation bar.
3. Search for the custom policy name you created earlier in this section, and choose the radio button next to the policy name to highlight it.
4. Choose the **Actions** drop-down and then choose **Delete**.
5. Type the name of the custom policy and then choose **Delete** to confirm deletion . This will automatically detach the policy from the `scorekeepRole` role.

Clean up

Follow these steps to delete the Scorekeep application resources:

> **Note**
>
> If you created and attached custom policies using the prior section of this tutorial, you must remove the policy from the `scorekeepRole` before deleting the CloudFormation stack.

AWS Management Console

Delete the sample application using the AWS Management Console

1. Open the CloudFormation console
2. Choose the radio button next to the `scorekeep` stack name in the list, and then choose **Delete**.

3. The CloudFormation stack is now being deleted. The stack status will be `DELETE_IN_PROGRESS` for a few minutes until all resources are deleted. The status will refresh periodically, or you can refresh the page.

AWS CLI

Delete the sample application using the AWS CLI

1. Enter the following AWS CLI command to delete the CloudFormation stack:

```
aws cloudformation delete-stack --stack-name scorekeep
```

2. Wait until the CloudFormation stack no longer exists, which will take about five minutes. Use the following AWS CLI command to check on the status:

```
aws cloudformation describe-stacks --stack-name scorekeep --query
 "Stacks[0].StackStatus"
```

Next steps

Learn more about X-Ray in the next chapter, AWS X-Ray concepts.

To instrument your own app, learn more about the X-Ray SDK for Java or one of the other X-Ray SDKs:

- **X-Ray SDK for Java** – AWS X-Ray SDK for Java
- **X-Ray SDK for Node.js** – AWS X-Ray SDK for Node.js
- **X-Ray SDK for .NET** – AWS X-Ray SDK for .NET

To run the X-Ray daemon locally or on AWS, see AWS X-Ray daemon.

To contribute to the sample application on GitHub, see eb-java-scorekeep.

Manually instrumenting AWS SDK clients

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The X-Ray SDK for Java automatically instruments all AWS SDK clients when you include the AWS SDK Instrumentor submodule in your build dependencies.

You can disable automatic client instrumentation by removing the Instrumentor submodule. This enables you to instrument some clients manually while ignoring others, or use different tracing handlers on different clients.

To illustrate support for instrumenting specific AWS SDK clients, the application passes a tracing handler to `AmazonDynamoDBClientBuilder` as a request handler in the user, game, and session model. This code change tells the SDK to instrument all calls to DynamoDB using those clients.

Example `src/main/java/scorekeep/SessionModel.java` – Manual AWS SDK client instrumentation

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.handlers.TracingHandler;

public class SessionModel {
  private AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
        .withRegion(Constants.REGION)
        .withRequestHandlers(new TracingHandler(AWSXRay.getGlobalRecorder()))
        .build();
  private DynamoDBMapper mapper = new DynamoDBMapper(client);
```

If you remove the AWS SDK Instrumentor submodule from project dependencies, only the manually instrumented AWS SDK clients appear in the trace map.

Creating additional subsegments

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

In the user model class, the application manually creates subsegments to group all downstream calls made within the `saveUser` function and adds metadata.

Example `src/main/java/scorekeep/UserModel.java` - Custom subsegments

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Subsegment;
...
    public void saveUser(User user) {
    // Wrap in subsegment
    Subsegment subsegment = AWSXRay.beginSubsegment("## UserModel.saveUser");
    try {
      mapper.save(user);
    } catch (Exception e) {
      subsegment.addException(e);
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
  }
```

Recording annotations, metadata, and user IDs

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive

updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

In the game model class, the application records Game objects in a metadata block each time it saves a game in DynamoDB. Separately, the application records game IDs in annotations for use with filter expressions.

Example `src/main/java/scorekeep/GameModel.java` – Annotations and metadata

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Segment;
import com.amazonaws.xray.entities.Subsegment;
...
  public void saveGame(Game game) throws SessionNotFoundException {
    // wrap in subsegment
    Subsegment subsegment = AWSXRay.beginSubsegment("## GameModel.saveGame");
    try {
      // check session
      String sessionId = game.getSession();
      if (sessionModel.loadSession(sessionId) == null ) {
        throw new SessionNotFoundException(sessionId);
      }
      Segment segment = AWSXRay.getCurrentSegment();
      subsegment.putMetadata("resources", "game", game);
      segment.putAnnotation("gameid", game.getId());
      mapper.save(game);
    } catch (Exception e) {
      subsegment.addException(e);
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
  }
```

In the move controller, the application records user IDs with `setUser`. User IDs are recorded in a separate field on segments and are indexed for use with search.

Example src/main/java/scorekeep/MoveController.java – User ID

```
import com.amazonaws.xray.AWSXRay;
...
  @RequestMapping(value="/{userId}", method=RequestMethod.POST)
  public Move newMove(@PathVariable String sessionId, @PathVariable String
 gameId, @PathVariable String userId, @RequestBody String move) throws
 SessionNotFoundException, GameNotFoundException, StateNotFoundException,
 RulesException {
    AWSXRay.getCurrentSegment().setUser(userId);
    return moveFactory.newMove(sessionId, gameId, userId, move);
  }
```

Instrumenting outgoing HTTP calls

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The user factory class shows how the application uses the X-Ray SDK for Java's version of `HTTPClientBuilder` to instrument outgoing HTTP calls.

Example `src/main/java/scorekeep/UserFactory.java` – HTTPClient instrumentation

```
import com.amazonaws.xray.proxies.apache.http.HttpClientBuilder;

  public String randomName() throws IOException {
    CloseableHttpClient httpclient = HttpClientBuilder.create().build();
    HttpGet httpGet = new HttpGet("http://uinames.com/api/");
    CloseableHttpResponse response = httpclient.execute(httpGet);
    try {
      HttpEntity entity = response.getEntity();
      InputStream inputStream = entity.getContent();
      ObjectMapper mapper = new ObjectMapper();
      Map<String, String> jsonMap = mapper.readValue(inputStream, Map.class);
      String name = jsonMap.get("name");
      EntityUtils.consume(entity);
      return name;
    } finally {
      response.close();
    }
  }
```

If you currently use `org.apache.http.impl.client.HttpClientBuilder`, you can simply swap out the import statement for that class with one for `com.amazonaws.xray.proxies.apache.http.HttpClientBuilder`.

Instrumenting calls to a PostgreSQL database

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

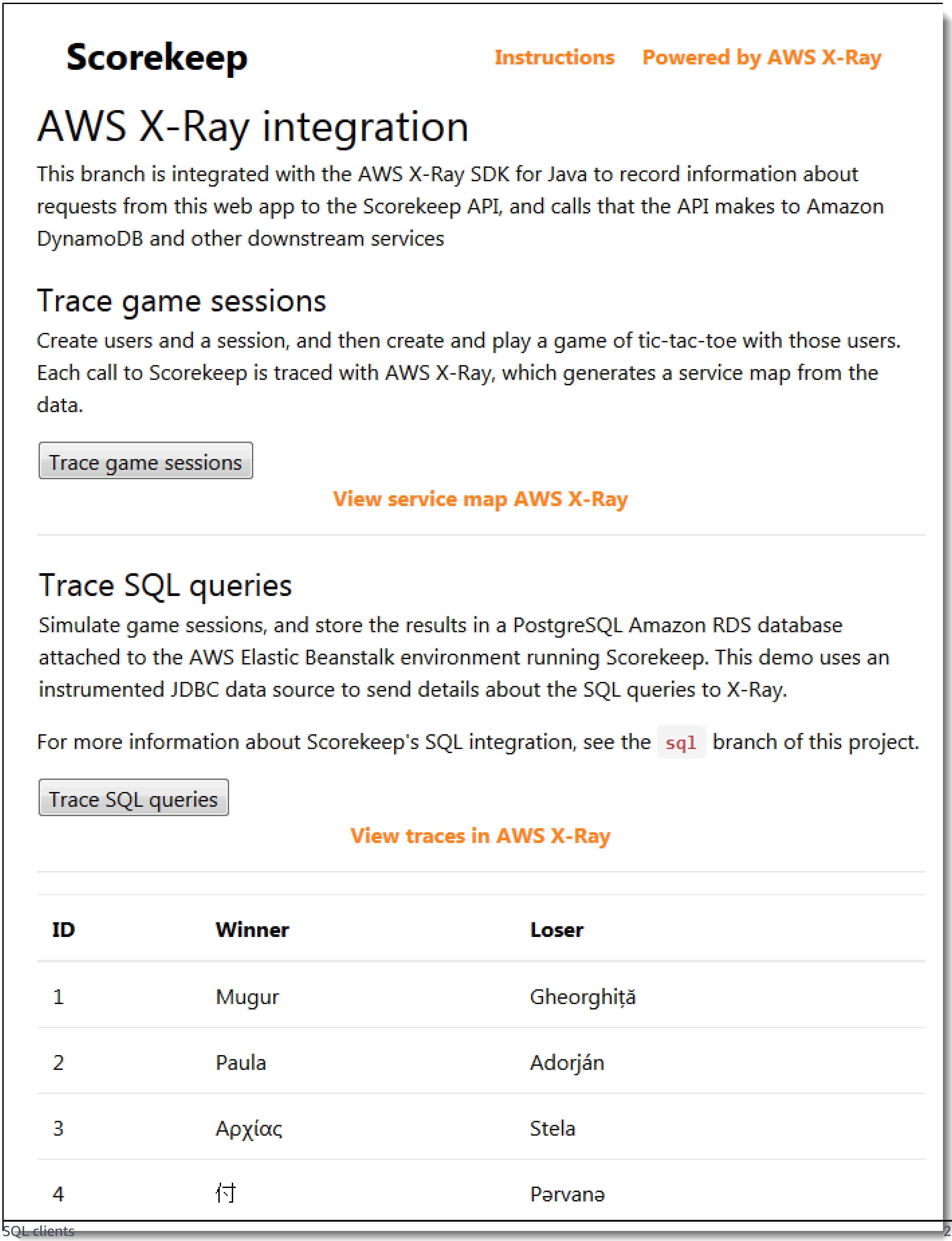
The `application-pgsql.properties` file adds the X-Ray PostgreSQL tracing interceptor to the data source created in `RdsWebConfig.java`.

Example `application-pgsql.properties` – PostgreSQL database instrumentation

```
spring.datasource.continue-on-error=true
spring.jpa.show-sql=false
spring.jpa.hibernate.ddl-auto=create-drop
spring.datasource.jdbc-interceptors=com.amazonaws.xray.sql.postgres.TracingInterceptor
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQL94Dialect
```

> **Note**
>
> See Configuring Databases with Elastic Beanstalk in the *AWS Elastic Beanstalk Developer Guide* for details on how to add a PostgreSQL database to the application environment.

The X-Ray demo page in the `xray` branch includes a demo that uses the instrumented data source to generate traces that show information about the SQL queries that it generates. Navigate to the `/#/xray` path in the running application or choose **Powered by AWS X-Ray** in the navigation bar to see the demo page.



AWS X-Ray integration

This branch is integrated with the AWS X-Ray SDK for Java to record information about requests from this web app to the Scorekeep API, and calls that the API makes to Amazon DynamoDB and other downstream services

Trace game sessions

Create users and a session, and then create and play a game of tic-tac-toe with those users. Each call to Scorekeep is traced with AWS X-Ray, which generates a service map from the data.

Trace SQL queries

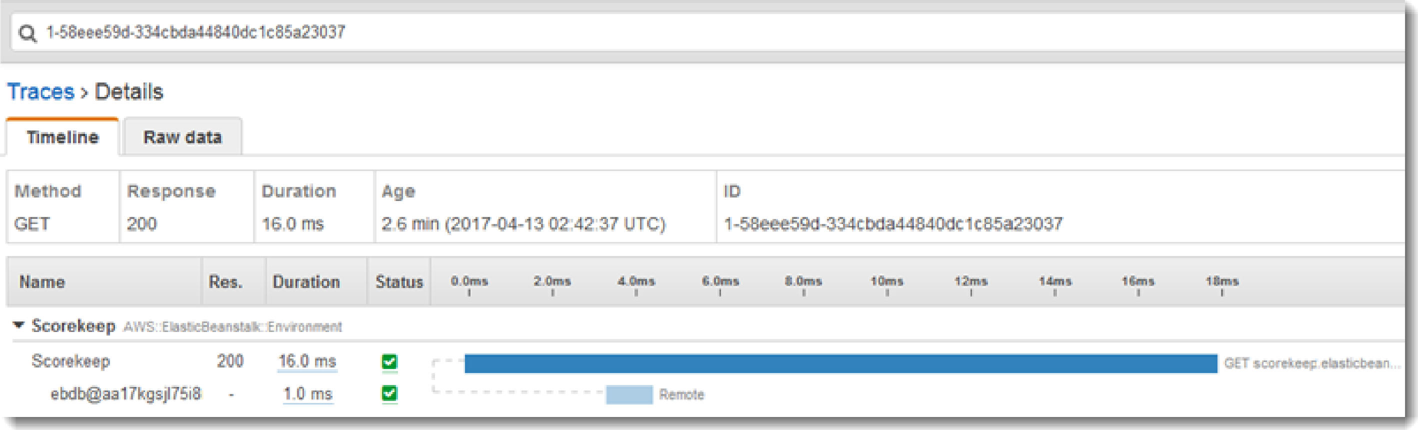
Simulate game sessions, and store the results in a PostgreSQL Amazon RDS database attached to the AWS Elastic Beanstalk environment running Scorekeep. This demo uses an instrumented JDBC data source to send details about the SQL queries to X-Ray.

For more information about Scorekeep's SQL integration, see the `sql` branch of this project.

ID	Winner	Loser
1	Mugur	Gheorghiță
2	Paula	Adorján
3	Αρχίας	Stela
4	付	Pərvanə

Choose **Trace SQL queries** to simulate game sessions and store the results in the attached database. Then, choose **View traces in AWS X-Ray** to see a filtered list of traces that hit the API's `/api/history` route.

Choose one of the traces from the list to see the timeline, including the SQL query.



Method	Response	Duration	Age	ID
GET	200	16.0 ms	2.6 min (2017-04-13 02:42:37 UTC)	1-58eee59d-334cbda44840dc1c85a23037

Name	Res.	Duration	Status
▾ Scorekeep AWS::ElasticBeanstalk::Environment			
Scorekeep	200	16.0 ms	✓
ebdb@aa17kgsjl75i8	-	1.0 ms	✓

Instrumenting AWS Lambda functions

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

Scorekeep uses two AWS Lambda functions. The first is a Node.js function from the `lambda` branch that generates random names for new users. When a user creates a session without entering a name, the application calls a function named `random-name` with the AWS SDK for Java. The X-Ray SDK for Java records information about the call to Lambda in a subsegment like any other call made with an instrumented AWS SDK client.

> **Note**
>
> Running the `random-name` Lambda function requires the creation of additional resources outside of the Elastic Beanstalk environment. See the readme for more information and instructions: AWS Lambda Integration.

The second function, `scorekeep-worker`, is a Python function that runs independently of the Scorekeep API. When a game ends, the API writes the session ID and game ID to an SQS queue. The worker function reads items from the queue, and calls the Scorekeep API to construct complete records of each game session for storage in Amazon S3.

Scorekeep includes AWS CloudFormation templates and scripts to create both functions. Because you need to bundle the X-Ray SDK with the function code, the templates create the functions without any code. When you deploy Scorekeep, a configuration file included in the `.ebextensions` folder creates a source bundle that includes the SDK, and updates the function code and configuration with the AWS Command Line Interface.

Functions

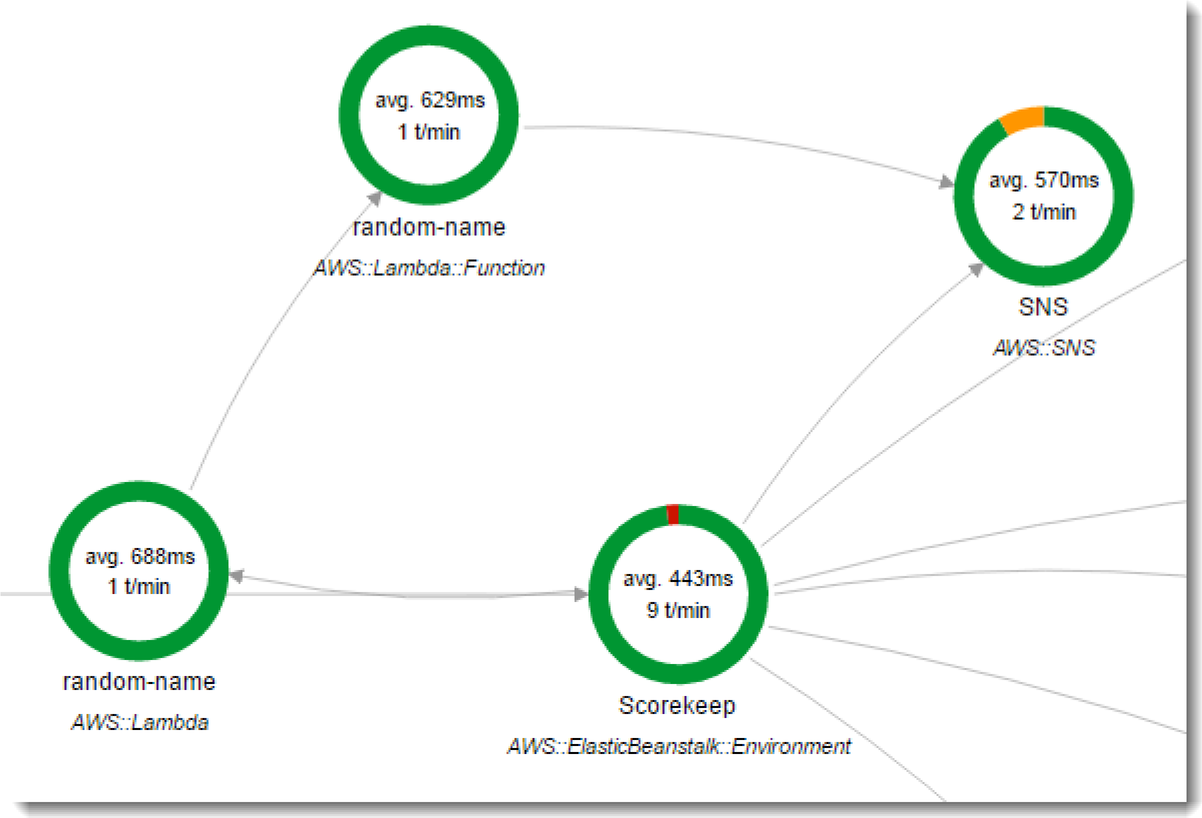
- Random name
- Worker

Random name

Scorekeep calls the random name function when a user starts a game session without signing in or specifying a user name. When Lambda processes the call to `random-name`, it reads the tracing header, which contains the trace ID and sampling decision written by the X-Ray SDK for Java.

For each sampled request, Lambda runs the X-Ray daemon and writes two segments. The first segment records information about the call to Lambda that invokes the function. This segment contains the same information as the subsegment recorded by Scorekeep, but from the Lambda point of view. The second segment represents the work that the function does.

Lambda passes the function segment to the X-Ray SDK through the function context. When you instrument a Lambda function, you don't use the SDK to create a segment for incoming requests. Lambda provides the segment, and you use the SDK to instrument clients and write subsegments.



The `random-name` function is implemented in Node.js. It uses the SDK for JavaScript in Node.js to send notifications with Amazon SNS, and the X-Ray SDK for Node.js to instrument the AWS SDK client. To write annotations, the function creates a custom subsegment with `AWSXRay.captureFunc`, and writes annotations in the instrumented function. In Lambda, you can't write annotations directly to the function segment, only to a subsegment that you create.

Example `function/index.js` -- Random name Lambda function

```
var AWSXRay = require('aws-xray-sdk-core');
var AWS = AWSXRay.captureAWS(require('aws-sdk'));

AWS.config.update({region: process.env.AWS_REGION});
var Chance = require('chance');

var myFunction = function(event, context, callback) {
  var sns = new AWS.SNS();
  var chance = new Chance();
  var userid = event.userid;
  var name = chance.first();

  AWSXRay.captureFunc('annotations', function(subsegment){
    subsegment.addAnnotation('Name', name);
    subsegment.addAnnotation('UserID', event.userid);
  });

  // Notify
  var params = {
    Message: 'Created randon name "' + name + '"" for user "' + userid + '".',
    Subject: 'New user: ' + name,
    TopicArn: process.env.TOPIC_ARN
  };
  sns.publish(params, function(err, data) {
    if (err) {
      console.log(err, err.stack);
      callback(err);
    }
    else {
      console.log(data);
      callback(null, {"name": name});
    }
  });
};

exports.handler = myFunction;
```

This function is created automatically when you deploy the sample application to Elastic Beanstalk. The `xray` branch includes a script to create a blank Lambda function. Configuration files in the `.ebextensions` folder build the function package with `npm install` during deployment, and then update the Lambda function with the AWS CLI.

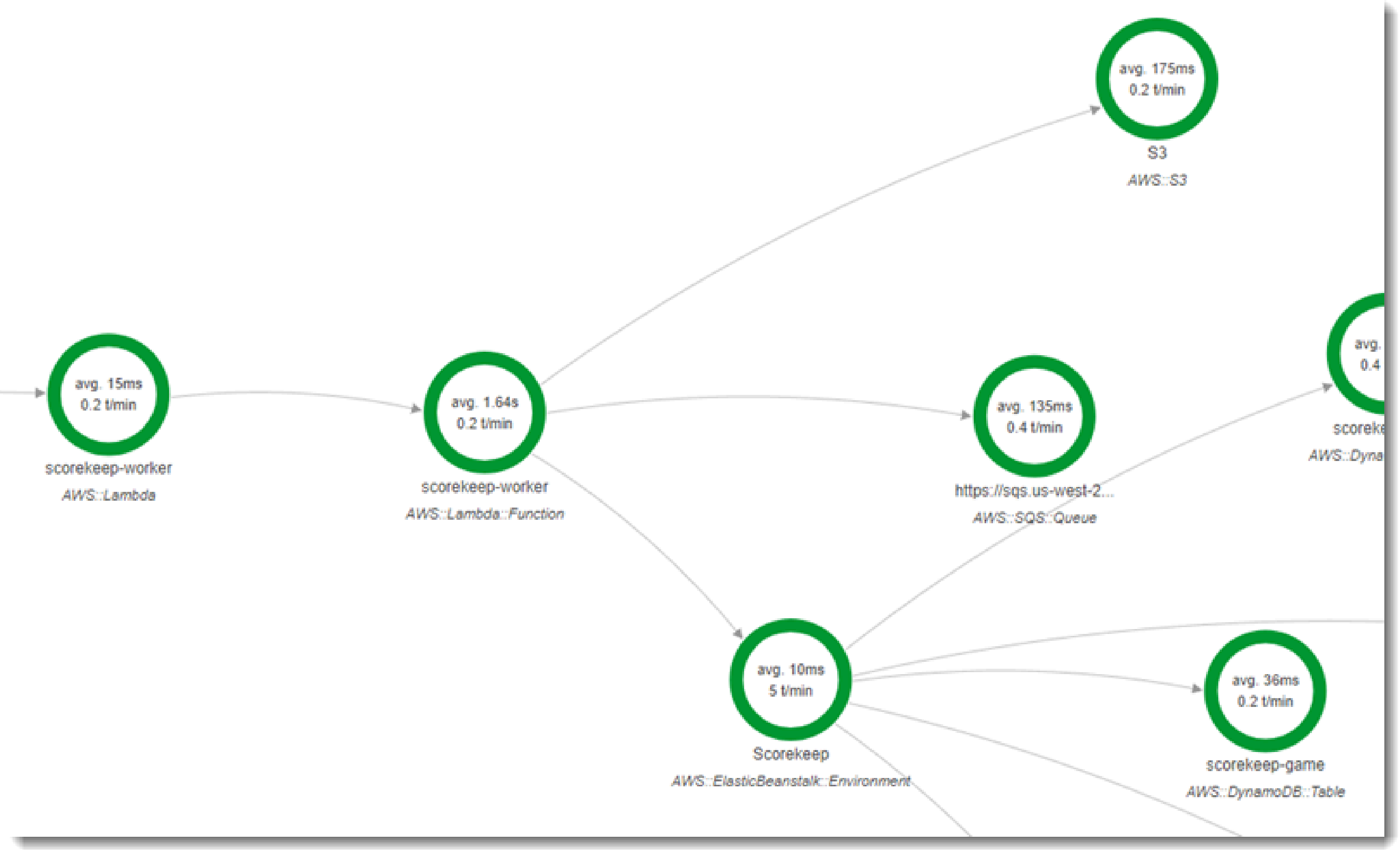
Worker

The instrumented worker function is provided in its own branch, `xray-worker`, as it cannot run unless you create the worker function and related resources first. See the branch readme for instructions.

The function is triggered by a bundled Amazon CloudWatch Events event every 5 minutes. When it runs, the function pulls an item from an Amazon SQS queue that Scorekeep manages. Each message contains information about a completed game.

The worker pulls the game record and documents from other tables that the game record references. For example, the game record in DynamoDB includes a list of moves that were executed during the game. The list does not contain the moves themselves, but rather IDs of moves that are stored in a separate table.

Sessions, and states are stored as references as well. This keeps the entries in the game table from being too large, but requires additional calls to get all of the information about the game. The worker dereferences all of these entries and constructs a complete record of the game as a single document in Amazon S3. When you want to do analytics on the data, you can run queries on it directly in Amazon S3 with Amazon Athena without running read-heavy data migrations to get your data out of DynamoDB.



The worker function has active tracing enabled in its configuration in AWS Lambda. Unlike the random name function, the worker does not receive a request from an instrumented application, so AWS Lambda doesn't receive a tracing header. With active tracing, Lambda creates the trace ID and makes sampling decisions.

The X-Ray SDK for Python is just a few lines at the top of the function that import the SDK and run its `patch_all` function to patch the AWS SDK for Python (Boto) and HTTclients that it uses to call Amazon SQS and Amazon S3. When the worker calls the Scorekeep API, the SDK adds the tracing header to the request to trace calls through the API.

Example _lambda/scorekeep-worker/scorekeep-worker.py -- Worker Lambda function

```
import os
import boto3
import json
import requests
import time
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

patch_all()
queue_url = os.environ['WORKER_QUEUE']

def lambda_handler(event, context):
    # Create SQS client
    sqs = boto3.client('sqs')
    s3client = boto3.client('s3')

    # Receive message from SQS queue
    response = sqs.receive_message(
        QueueUrl=queue_url,
        AttributeNames=[
            'SentTimestamp'
        ],
        MaxNumberOfMessages=1,
        MessageAttributeNames=[
            'All'
        ],
        VisibilityTimeout=0,
        WaitTimeSeconds=0
    )
   ...
```

Instrumenting startup code

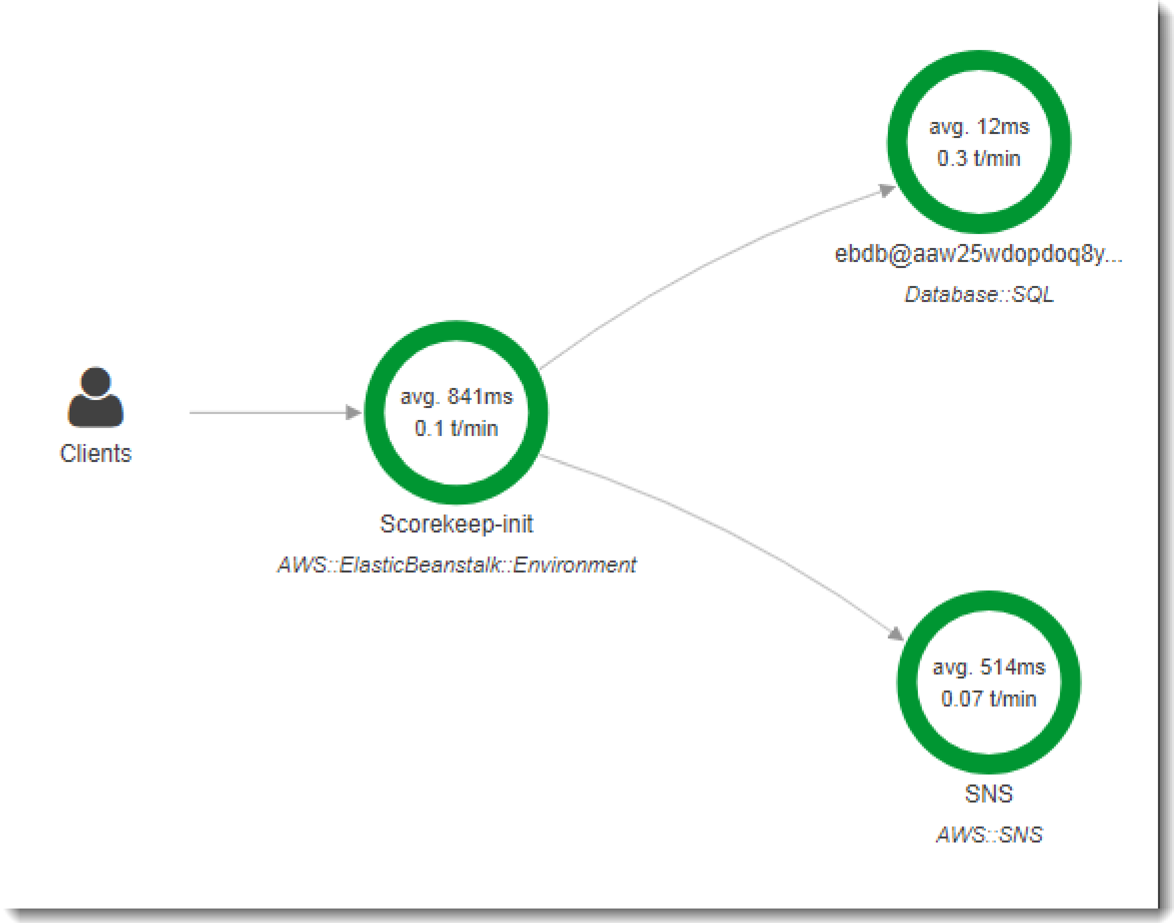
Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more

information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The X-Ray SDK for Java automatically creates segments for incoming requests. As long as a request is in scope, you can use instrumented clients and record subsegments without issue. If you try to use an instrumented client in startup code, though, you'll get a SegmentNotFoundException.

Startup code runs outside of the standard request/response flow of a web application, so you need to create segments manually to instrument it. Scorekeep shows the instrumentation of startup code in its `WebConfig` files. Scorekeep calls an SQL database and Amazon SNS during startup.



The default `WebConfig` class creates an Amazon SNS subscription for notifications. To provide a segment for the X-Ray SDK to write to when the Amazon SNS client is used, Scorekeep calls `beginSegment` and `endSegment` on the global recorder.

Example `src/main/java/scorekeep/WebConfig.java` – Instrumented AWS SDK client in startup code

```
AWSXRay.beginSegment("Scorekeep-init");
if ( System.getenv("NOTIFICATION_EMAIL") != null ){
  try { Sns.createSubscription(); }
  catch (Exception e ) {
    logger.warn("Failed to create subscription for email "+
 System.getenv("NOTIFICATION_EMAIL"));
  }
}
AWSXRay.endSegment();
```

In `RdsWebConfig`, which Scorekeep uses when an Amazon RDS database is connected, the configuration also creates a segment for the SQL client that Hibernate uses when it applies the database schema during startup.

Example `src/main/java/scorekeep/RdsWebConfig.java` – Instrumented SQL database client in startup code

```
@PostConstruct
public void schemaExport() {
  EntityManagerFactoryImpl entityManagerFactoryImpl = (EntityManagerFactoryImpl)
 localContainerEntityManagerFactoryBean.getNativeEntityManagerFactory();
  SessionFactoryImplementor sessionFactoryImplementor =
 entityManagerFactoryImpl.getSessionFactory();
  StandardServiceRegistry standardServiceRegistry =
 sessionFactoryImplementor.getSessionFactoryOptions().getServiceRegistry();
  MetadataSources metadataSources = new MetadataSources(new
 BootstrapServiceRegistryBuilder().build());
  metadataSources.addAnnotatedClass(GameHistory.class);
  MetadataImplementor metadataImplementor = (MetadataImplementor)
 metadataSources.buildMetadata(standardServiceRegistry);
  SchemaExport schemaExport = new SchemaExport(standardServiceRegistry,
 metadataImplementor);

  AWSXRay.beginSegment("Scorekeep-init");
  schemaExport.create(true, true);
  AWSXRay.endSegment();
}
```

`SchemaExport` runs automatically and uses an SQL client. Since the client is instrumented, Scorekeep must override the default implementation and provide a segment for the SDK to use when the client is invoked.

Instrumenting scripts

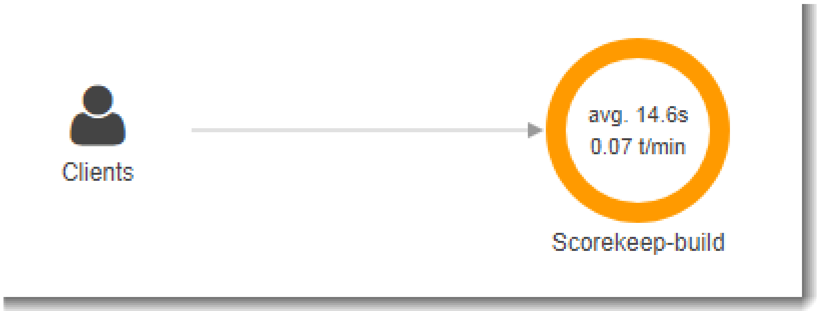
> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

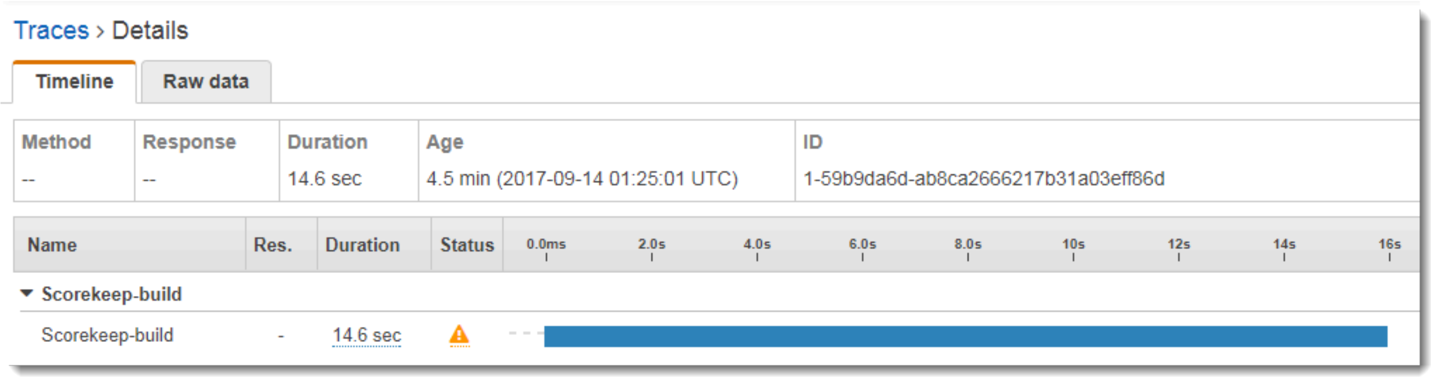
You can also instrument code that isn't part of your application. When the X-Ray daemon is running, it will relay any segments that it receives to X-Ray, even if they are not generated by the X-Ray SDK. Scorekeep uses its own scripts to instrument the build that compiles the application during deployment.

Example `bin/build.sh` – Instrumented build script

```
SEGMENT=$(python bin/xray_start.py)
gradle build --quiet --stacktrace &> /var/log/gradle.log; GRADLE_RETURN=$?
if (( GRADLE_RETURN != 0 )); then
  echo "Gradle failed with exit status $GRADLE_RETURN" >&2
  python bin/xray_error.py "$SEGMENT" "$(cat /var/log/gradle.log)"
  exit 1
fi
python bin/xray_success.py "$SEGMENT"
```

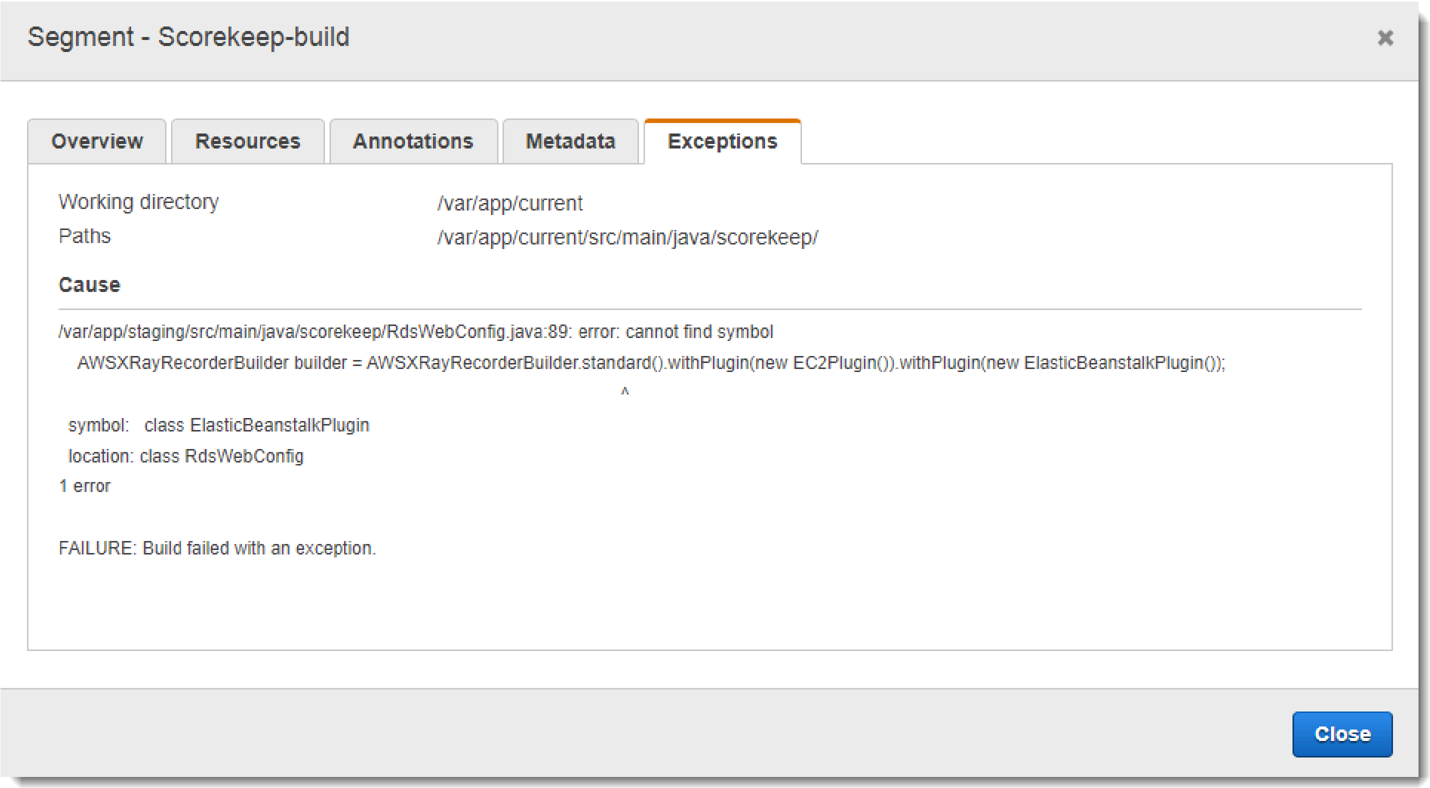
`xray_start.py`, `xray_error.py` and `xray_success.py` are simple Python scripts that construct segment objects, convert them to JSON documents, and send them to the daemon over UDP. If the Gradle build fails, you can find the error message by clicking on the **scorekeep-build** node in the X-Ray console trace map.





Method	Response	Duration	Age	ID
--	--	14.6 sec	4.5 min (2017-09-14 01:25:01 UTC)	1-59b9da6d-ab8ca2666217b31a03eff86d

Name	Res.	Duration	Status
▼ Scorekeep-build			
Scorekeep-build	-	14.6 sec	⚠



Working directory /var/app/current

Paths /var/app/current/src/main/java/scorekeep/

Cause

```
/var/app/staging/src/main/java/scorekeep/RdsWebConfig.java:89: error: cannot find symbol
    AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder.standard().withPlugin(new EC2Plugin()).withPlugin(new ElasticBeanstalkPlugin());
                                                                                                                  ^
  symbol:   class ElasticBeanstalkPlugin
  location: class RdsWebConfig
1 error

FAILURE: Build failed with an exception.
```

Instrumenting a web app client

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

In the `xray-cognito` branch, Scorekeep uses Amazon Cognito to enable users to create an account and sign in with it to retrieve their user information from an Amazon Cognito user pool. When a user signs in, Scorekeep uses an Amazon Cognito identity pool to get temporary AWS credentials for use with the AWS SDK for JavaScript.

The identity pool is configured to let signed-in users write trace data to AWS X-Ray. The web app uses these credentials to record the signed-in user's ID, the browser path, and the client's view of calls to the Scorekeep API.

Most of the work is done in a service class named `xray`. This service class provides methods for generating the required identifiers, creating in-progress segments, finalizing segments, and sending segment documents to the X-Ray API.

Example `public/xray.js` – Record and upload segments

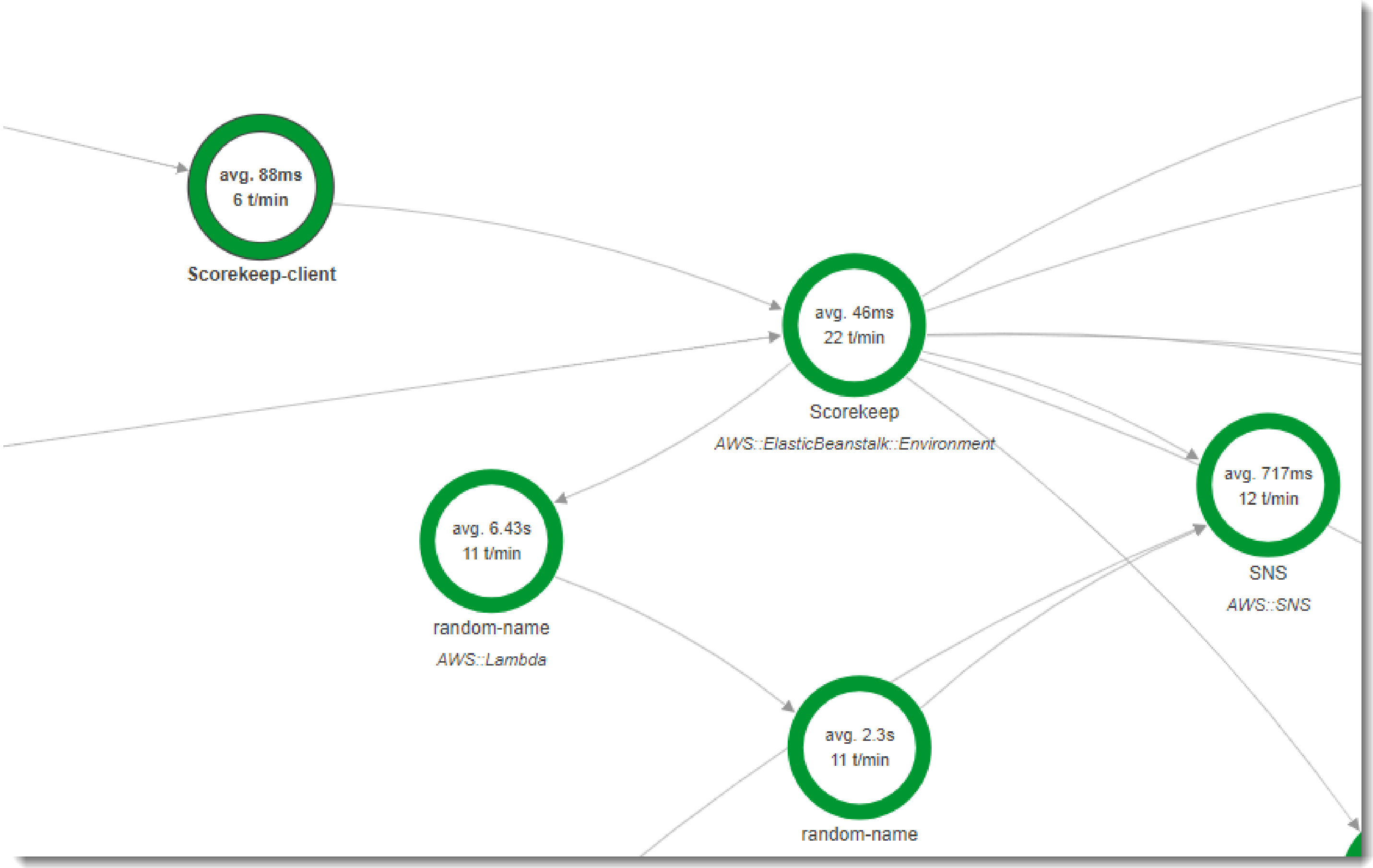
```
...
  service.beginSegment = function() {
    var segment = {};
    var traceId = '1-' + service.getHexTime() + '-' + service.getHexId(24);

    var id = service.getHexId(16);
    var startTime = service.getEpochTime();

    segment.trace_id = traceId;
    segment.id = id;
    segment.start_time = startTime;
    segment.name = 'Scorekeep-client';
    segment.in_progress = true;
    segment.user =  sessionStorage['userid'];
    segment.http = {
      request: {
        url: window.location.href
      }
    };

    var documents = [];
    documents[0] = JSON.stringify(segment);
    service.putDocuments(documents);
    return segment;
  }

  service.endSegment = function(segment) {
    var endTime = service.getEpochTime();
    segment.end_time = endTime;
    segment.in_progress = false;
    var documents = [];
    documents[0] = JSON.stringify(segment);
    service.putDocuments(documents);
  }

  service.putDocuments = function(documents) {
    var xray = new AWS.XRay();
    var params = {
      TraceSegmentDocuments: documents
    };
    xray.putTraceSegments(params, function(err, data) {
      if (err) {
        console.log(err, err.stack);
      } else {
        console.log(data);
      }
    })
  }
```

These methods are called in header and `transformResponse` functions in the resource services that the web app uses to call the Scorekeep API. To include the client segment in the same trace as the segment that the API generates, the web app must include the trace ID and segment ID in a tracing header (`X-Amzn-Trace-Id`) that the X-Ray SDK can read. When the instrumented Java application receives a request with this header, the X-Ray SDK for Java uses the same trace ID and makes the segment from the web app client the parent of its segment.

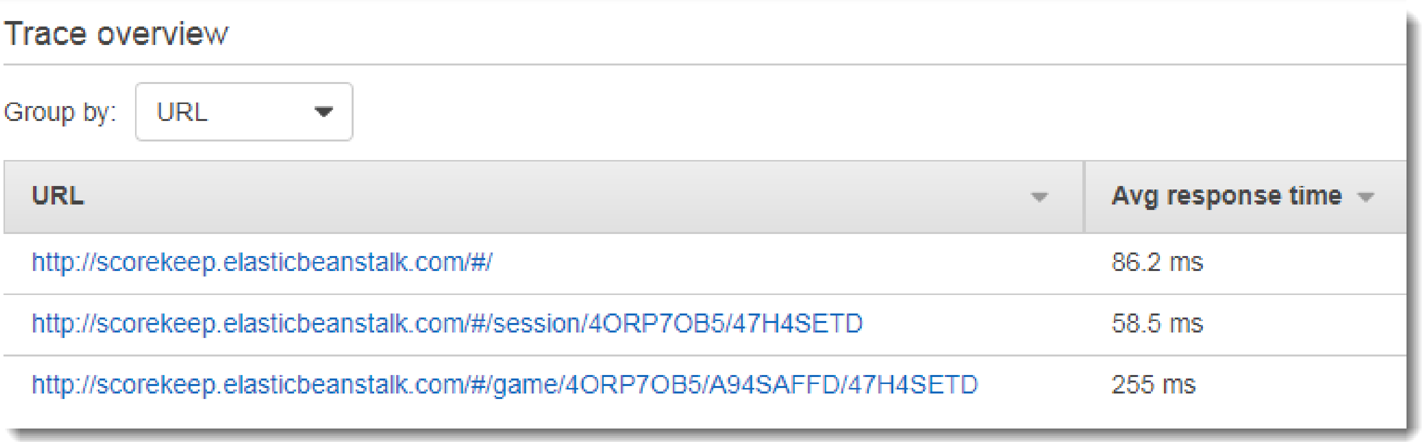
Example public/app/services.js – Recording segments for angular resource calls and writing tracing headers

```
var module = angular.module('scorekeep');
module.factory('SessionService', function($resource, api, XRay) {
  return $resource(api + 'session/:id', { id: '@_id' }, {
    segment: {},
    get: {
      method: 'GET',
      headers: {
        'X-Amzn-Trace-Id': function(config) {
          segment = XRay.beginSegment();
          return XRay.getTraceHeader(segment);
        }
      },
      transformResponse: function(data) {
        XRay.endSegment(segment);
        return angular.fromJson(data);
      },
    },
...
```

The resulting trace map includes a node for the web app client.



Traces that include segments from the web app show the URL that the user sees in the browser (paths starting with `/#/`). Without client instrumentation, you only get the URL of the API resource that the web app calls (paths starting with `/api/`).



Trace overview

Group by: URL ▾

URL	Avg response time
http://scorekeep.elasticbeanstalk.com/#/	86.2 ms
http://scorekeep.elasticbeanstalk.com/#/session/4ORP7OB5/47H4SETD	58.5 ms
http://scorekeep.elasticbeanstalk.com/#/game/4ORP7OB5/A94SAFFD/47H4SETD	255 ms

Using instrumented clients in worker threads

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

Scorekeep uses a worker thread to publish a notification to Amazon SNS when a user wins a game. Publishing the notification takes longer than the rest of the request operations combined, and doesn't affect the client or user. Therefore, performing the task asynchronously is a good way to improve response time.

However, the X-Ray SDK for Java doesn't know which segment was active when the thread is created. As a result, when you try to use the instrumented AWS SDK for Java client within the thread, it throws a `SegmentNotFoundException`, crashing the thread.

Example Web-1.error.log

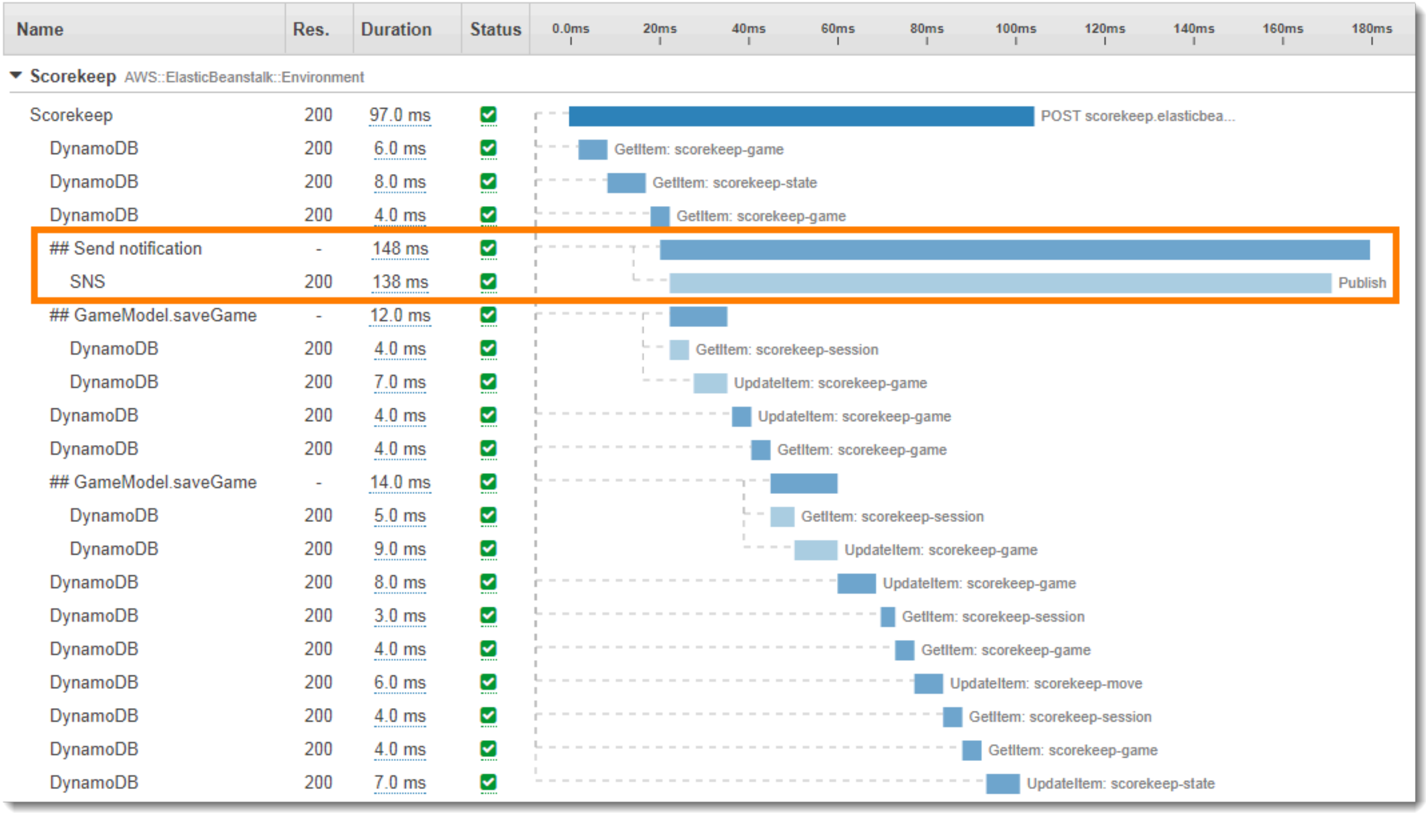
```
Exception in thread "Thread-2" com.amazonaws.xray.exceptions.SegmentNotFoundException:
 Failed to begin subsegment named 'AmazonSNS': segment cannot be found.
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at
 sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
        at
 sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.ja
...
```

To fix this, the application uses `GetTraceEntity` to get a reference to the segment in the main thread, and `Entity.run()` to safely run the worker thread code with access to the segment's context.

Example `src/main/java/scorekeep/MoveFactory.java` – Passing trace context to a worker thread

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorder;
import com.amazonaws.xray.entities.Entity;
import com.amazonaws.xray.entities.Segment;
import com.amazonaws.xray.entities.Subsegment;
...
      Entity segment = recorder.getTraceEntity();
      Thread comm = new Thread() {
        public void run() {
         segment.run(() -> {
            Subsegment subsegment = AWSXRay.beginSubsegment("## Send notification");
            Sns.sendNotification("Scorekeep game completed", "Winner: " + userId);
            AWSXRay.endSubsegment();
          }
        }
```

Because the request is now resolved before the call to Amazon SNS, the application creates a separate subsegment for the thread. This prevents the X-Ray SDK from closing the segment before it records the response from Amazon SNS. If no subsegment is open when Scorekeep resolved the request, the response from Amazon SNS could be lost.



Name	Res.	Duration	Status	0.0ms 20ms 40ms 60ms 80ms 100ms 120ms 140ms 160ms 180ms
▾ Scorekeep AWS::ElasticBeanstalk::Environment				
Scorekeep	200	97.0 ms	✓	POST scorekeep.elasticbea...
DynamoDB	200	6.0 ms	✓	GetItem: scorekeep-game
DynamoDB	200	8.0 ms	✓	GetItem: scorekeep-state
DynamoDB	200	4.0 ms	✓	GetItem: scorekeep-game
## Send notification	-	148 ms	✓	
SNS	200	138 ms	✓	Publish
## GameModel.saveGame	-	12.0 ms	✓	
DynamoDB	200	4.0 ms	✓	GetItem: scorekeep-session
DynamoDB	200	7.0 ms	✓	UpdateItem: scorekeep-game
DynamoDB	200	4.0 ms	✓	UpdateItem: scorekeep-game
DynamoDB	200	4.0 ms	✓	GetItem: scorekeep-game
## GameModel.saveGame	-	14.0 ms	✓	
DynamoDB	200	5.0 ms	✓	GetItem: scorekeep-session
DynamoDB	200	9.0 ms	✓	UpdateItem: scorekeep-game
DynamoDB	200	8.0 ms	✓	UpdateItem: scorekeep-game
DynamoDB	200	3.0 ms	✓	GetItem: scorekeep-session
DynamoDB	200	4.0 ms	✓	GetItem: scorekeep-game
DynamoDB	200	6.0 ms	✓	UpdateItem: scorekeep-move
DynamoDB	200	4.0 ms	✓	GetItem: scorekeep-session
DynamoDB	200	4.0 ms	✓	GetItem: scorekeep-game
DynamoDB	200	7.0 ms	✓	UpdateItem: scorekeep-state

See Passing segment context between threads in a multithreaded application for more information about multithreading.

AWS X-Ray daemon

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

Note

You can now use the CloudWatch agent to collect metrics, logs and traces from Amazon EC2 instances and on-premise servers. CloudWatch agent version 1.300025.0 and later can collect traces from OpenTelemetry or X-Ray client SDKs, and send them to X-Ray. Using the CloudWatch agent instead of the AWS Distro for OpenTelemetry (ADOT) Collector or X-Ray daemon to collect traces can help you reduce the number of agents that you manage. See the CloudWatch agent topic in the CloudWatch User Guide for more information.

The AWS X-Ray daemon is a software application that listens for traffic on UDP port 2000, gathers raw segment data, and relays it to the AWS X-Ray API. The daemon works in conjunction with the AWS X-Ray SDKs and must be running so that data sent by the SDKs can reach the X-Ray service. The X-Ray daemon is an open source project. You can follow the project and submit issues and pull requests on GitHub: github.com/aws/aws-xray-daemon

On AWS Lambda and AWS Elastic Beanstalk, use those services' integration with X-Ray to run the daemon. Lambda runs the daemon automatically any time a function is invoked for a sampled request. On Elastic Beanstalk, use the `XRayEnabled` configuration option to run the daemon on the instances in your environment. For more information, see

To run the X-Ray daemon locally, on-premises, or on other AWS services, download it, run it, and then give it permission to upload segment documents to X-Ray.

Downloading the daemon

You can download the daemon from Amazon S3, Amazon ECR, or Docker Hub, and then run it locally, or install it on an Amazon EC2 instance on launch.

Amazon S3

X-Ray daemon installers and executables

- **Linux (executable)** – `aws-xray-daemon-linux-3.x.zip` (sig)
- **Linux (RPM installer)** – `aws-xray-daemon-3.x.rpm`
- **Linux (DEB installer)** – `aws-xray-daemon-3.x.deb`
- **Linux (ARM64, executable)** – `aws-xray-daemon-linux-arm64-3.x.zip` (sig)
- **Linux (ARM64, RPM installer)** – `aws-xray-daemon-arm64-3.x.rpm`
- **Linux (ARM64, DEB installer)** – `aws-xray-daemon-arm64-3.x.deb`
- **OS X (executable)** – `aws-xray-daemon-macos-3.x.zip` (sig)
- **Windows (executable)** – `aws-xray-daemon-windows-process-3.x.zip` (sig)
- **Windows (service)** – `aws-xray-daemon-windows-service-3.x.zip` (sig)

These links always point to the latest 3.x release of the daemon. To download a specific release, do the following:

- If you want to download a release prior to version `3.3.0`, replace `3.x` with the version number. For example, `2.1.0`. Prior to version `3.3.0`, the only available architecture is `arm64`. For example, `2.1.0` and `arm64`.
- If you want to download a release after version `3.3.0`, replace `3.x` with the version number and `arch` with the architecture type.

X-Ray assets are replicated to buckets in every supported region. To use the bucket closest to you or your AWS resources, replace the region in the above links with your region.

```
https://s3.us-west-2.amazonaws.com/aws-xray-assets.us-west-2/xray-daemon/aws-xray-
daemon-3.x.rpm
```

Amazon ECR

As of version 3.2.0 the daemon can be found on Amazon ECR. Before pulling an image you should authenticate your docker client to the Amazon ECR public registry.

Pull the latest released 3.x version tag by running the following command:

```
docker pull public.ecr.aws/xray/aws-xray-daemon:3.x
```

Prior or alpha releases can be downloaded by replacing `3.x` with `alpha` or a specific version number. It is not recommended to use a daemon image with an alpha tag in a production environment.

Docker Hub

The daemon can be found on Docker Hub. To download the latest released 3.x version, run the following command:

```
docker pull amazon/aws-xray-daemon:3.x
```

Prior releases of the daemon can be released by replacing `3.x` with the desired version.

Verifying the daemon archive's signature

GPG signature files are included for daemon assets compressed in ZIP archives. The public key is here: `aws-xray.gpg`.

You can use the public key to verify that the daemon's ZIP archive is original and unmodified. First, import the public key with GnuPG.

To import the public key

1. Download the public key.

   ```
   $ BUCKETURL=https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2
   $ wget $BUCKETURL/xray-daemon/aws-xray.gpg
   ```

2. Import the public key into your keyring.

   ```
   $ gpg --import aws-xray.gpg
   ```

```
gpg: /Users/me/.gnupg/trustdb.gpg: trustdb created
gpg: key 7BFE036BFE6157D3: public key "AWS X-Ray <aws-xray@amazon.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1
```

Use the imported key to verify the signature of the daemon's ZIP archive.

To verify an archive's signature

1. Download the archive and signature file.

   ```
   $ BUCKETURL=https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2
   $ wget $BUCKETURL/xray-daemon/aws-xray-daemon-linux-3.x.zip
   $ wget $BUCKETURL/xray-daemon/aws-xray-daemon-linux-3.x.zip.sig
   ```

2. Run `gpg --verify` to verify the signature.

   ```
   $ gpg --verify aws-xray-daemon-linux-3.x.zip.sig aws-xray-daemon-linux-3.x.zip
   gpg: Signature made Wed 19 Apr 2017 05:06:31 AM UTC using RSA key ID FE6157D3
   gpg: Good signature from "AWS X-Ray <aws-xray@amazon.com>"
   gpg: WARNING: This key is not certified with a trusted signature!
   gpg:          There is no indication that the signature belongs to the owner.
   Primary key fingerprint: EA6D 9271 FBF3 6990 277F  4B87 7BFE 036B FE61 57D3
   ```

Note the warning about trust. A key is only trusted if you or someone you trust has signed it. This does not mean that the signature is invalid, only that you have not verified the public key.

Running the daemon

Run the daemon locally from the command line. Use the `-o` option to run in local mode, and `-n` to set the region.

```
~/Downloads$ ./xray -o -n us-east-2
```

For detailed platform-specific instructions, see the following topics:

- **Linux (local)** – Running the X-Ray daemon on Linux
- **Windows (local)** – Running the X-Ray daemon on Windows

- **Elastic Beanstalk** – Running the X-Ray daemon on AWS Elastic Beanstalk
- **Amazon EC2** – Running the X-Ray daemon on Amazon EC2
- **Amazon ECS** – Running the X-Ray daemon on Amazon ECS

You can customize the daemon's behavior further by using command line options or a configuration file. See Configuring the AWS X-Ray daemon for details.

Giving the daemon permission to send data to X-Ray

The X-Ray daemon uses the AWS SDK to upload trace data to X-Ray, and it needs AWS credentials with permission to do that.

On Amazon EC2, the daemon uses the instance's instance profile role automatically. For information about credentials required to run the daemon locally, see running your application locally.

If you specify credentials in more than one location (credentials file, instance profile, or environment variables), the SDK provider chain determines which credentials are used. For more information about providing credentials to the SDK, see Specifying Credentials in the *AWS SDK for Go Developer Guide*.

The IAM role or user that the daemon's credentials belong to must have permission to write data to the service on your behalf.

- To use the daemon on Amazon EC2, create a new instance profile role or add the managed policy to an existing one.
- To use the daemon on Elastic Beanstalk, add the managed policy to the Elastic Beanstalk default instance profile role.
- To run the daemon locally, see running your application locally.

For more information, see Identity and access management for AWS X-Ray.

X-Ray daemon logs

The daemon outputs information about its current configuration and segments that it sends to AWS X-Ray.

```
2016-11-24T06:07:06Z [Info] Initializing AWS X-Ray daemon 2.1.0
2016-11-24T06:07:06Z [Info] Using memory limit of 49 MB
2016-11-24T06:07:06Z [Info] 313 segment buffers allocated
2016-11-24T06:07:08Z [Info] Successfully sent batch of 1 segments (0.123 seconds)
2016-11-24T06:07:09Z [Info] Successfully sent batch of 1 segments (0.006 seconds)
```

By default, the daemon outputs logs to STDOUT. If you run the daemon in the background, use the `--log-file` command line option or a configuration file to set the log file path. You can also set the log level and disable log rotation. See Configuring the AWS X-Ray daemon for instructions.

On Elastic Beanstalk, the platform sets the location of the daemon logs. See Running the X-Ray daemon on AWS Elastic Beanstalk for details.

Configuring the AWS X-Ray daemon

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can use command line options or a configuration file to customize the X-Ray daemon's behavior. Most options are available using both methods, but some are only available in configuration files and some only at the command line.

To get started, the only option that you need to know is `-n` or `--region`, which you use to set the region that the daemon uses to send trace data to X-Ray.

```
~/xray-daemon$ ./xray -n us-east-2
```

If you are running the daemon locally, that is, not on Amazon EC2, you can add the `-o` option to skip checking for instance profile credentials so the daemon will become ready more quickly.

```
~/xray-daemon$ ./xray -o -n us-east-2
```

The rest of the command line options let you configure logging, listen on a different port, limit the amount of memory that the daemon can use, or assume a role to send trace data to a different account.

You can pass a configuration file to the daemon to access advanced configuration options and do things like limit the number of concurrent calls to X-Ray, disable log rotation, and send traffic to a proxy.

Sections

- Supported environment variables
- Using command line options
- Using a configuration file

Supported environment variables

The X-Ray daemon supports the following environment variables:

- `AWS_REGION` – Specifies the AWS Region of the X-Ray service endpoint.
- `HTTPS_PROXY` – Specifies a proxy address for the daemon to upload segments through. This can be either the DNS domain names or IP addresses and port numbers used by your proxy servers.

Using command line options

Pass these options to the daemon when you run it locally or with a user data script.

Command Line Options

- `-b`, `--bind` – Listen for segment documents on a different UDP port.

  ```
  --bind "127.0.0.1:3000"
  ```

  Default – `2000`.
- `-t`, `--bind-tcp` – Listen for calls to the X-Ray service on a different TCP port.

  ```
  -bind-tcp "127.0.0.1:3000"
  ```

  Default – `2000`.

- `-c`, `--config` – Load a configuration file from the specified path.

```
--config "/home/ec2-user/xray-daemon.yaml"
```

- `-f`, `--log-file` – Output logs to the specified file path.

```
--log-file "/var/log/xray-daemon.log"
```

- `-l`, `--log-level` – Log level, from most verbose to least: dev, debug, info, warn, error, prod.

```
--log-level warn
```

  Default – `prod`

- `-m`, `--buffer-memory` – Change the amount of memory in MB that buffers can use (minimum 3).

```
--buffer-memory 50
```

  Default – 1% of available memory.

- `-o`, `--local-mode` – Don't check for EC2 instance metadata.
- `-r`, `--role-arn` – Assume the specified IAM role to upload segments to a different account.

```
--role-arn "arn:aws:iam::123456789012:role/xray-cross-account"
```

- `-a`, `--resource-arn` – Amazon Resource Name (ARN) of the AWS resource running the daemon.
- `-p`, `--proxy-address` – Upload segments to AWS X-Ray through a proxy. The proxy server's protocol must be specified.

```
--proxy-address "http://192.0.2.0:3000"
```

- `-n`, `--region` – Send segments to X-Ray service in a specific region.
- `-v`, `--version` – Show AWS X-Ray daemon version.
- `-h`, `--help` – Show the help screen.

Using a configuration file

You can also use a YAML format file to configure the daemon. Pass the configuration file to the daemon by using the `-c` option.

```
~$ ./xray -c ~/xray-daemon.yaml
```

Configuration file options

- `TotalBufferSizeMB` – Maximum buffer size in MB (minimum 3). Choose 0 to use 1% of host memory.
- `Concurrency` – Maximum number of concurrent calls to AWS X-Ray to upload segment documents.
- `Region` – Send segments to AWS X-Ray service in a specific region.
- `Socket` – Configure the daemon's binding.
  - `UDPAddress` – Change the port on which the daemon listens.
  - `TCPAddress` – Listen for calls to the X-Ray service on a different TCP port.
- `Logging` – Configure logging behavior.
  - `LogRotation` – Set to `false` to disable log rotation.
  - `LogLevel` – Change the log level, from most verbose to least: `dev`, `debug`, `info` or `prod`, `warn`, `error`, `prod`. The default is `prod`, which is equivalent to `info`.
  - `LogPath` – Output logs to the specified file path.
- `LocalMode` – Set to `true` to skip checking for EC2 instance metadata.
- `ResourceARN` – Amazon Resource Name (ARN) of the AWS resource running the daemon.
- `RoleARN` – Assume the specified IAM role to upload segments to a different account.
- `ProxyAddress` – Upload segments to AWS X-Ray through a proxy.
- `Endpoint` – Change the X-Ray service endpoint to which the daemon sends segment documents.
- `NoVerifySSL` – Disable TLS certificate verification.
- `Version` – Daemon configuration file format version. The file format version is a **required** field.

Example Xray-daemon.yaml

This configuration file changes the daemon's listening port to 3000, turns off checks for instance metadata, sets a role to use for uploading segments, and changes region and logging options.

```
Socket:
  UDPAddress: "127.0.0.1:3000"
  TCPAddress: "127.0.0.1:3000"
Region: "us-west-2"
Logging:
  LogLevel: "warn"
  LogPath: "/var/log/xray-daemon.log"
LocalMode: true
RoleARN: "arn:aws:iam::123456789012:role/xray-cross-account"
Version: 2
```

Running the X-Ray daemon locally

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can run the AWS X-Ray daemon locally on Linux, MacOS, Windows, or in a Docker container. Run the daemon to relay trace data to X-Ray when you are developing and testing your instrumented application. Download and extract the daemon by using the instructions here.

When running locally, the daemon can read credentials from an AWS SDK credentials file (`.aws/credentials` in your user directory) or from environment variables. For more information, see Giving the daemon permission to send data to X-Ray.

The daemon listens for UDP data on port 2000. You can change the port and other options by using a configuration file and command line options. For more information, see Configuring the AWS X-Ray daemon.

Running the X-Ray daemon on Linux

You can run the daemon executable from the command line. Use the `-o` option to run in local mode, and `-n` to set the region.

```
~/xray-daemon$ ./xray -o -n us-east-2
```

To run the daemon in the background, use &.

```
~/xray-daemon$ ./xray -o -n us-east-2 &
```

Terminate a daemon process running in the background with `pkill`.

```
~$ pkill xray
```

Running the X-Ray daemon in a Docker container

To run the daemon locally in a Docker container, save the following text to a file named `Dockerfile`. Download the complete example image on Amazon ECR. See downloading the daemon for more information.

Example Dockerfile – Amazon Linux

```
FROM amazonlinux
RUN yum install -y unzip
RUN curl -o daemon.zip https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/
xray-daemon/aws-xray-daemon-linux-3.x.zip
RUN unzip daemon.zip && cp xray /usr/bin/xray
ENTRYPOINT ["/usr/bin/xray", "-t", "0.0.0.0:2000", "-b", "0.0.0.0:2000"]
EXPOSE 2000/udp
EXPOSE 2000/tcp
```

Build the container image with `docker build`.

```
~/xray-daemon$ docker build -t xray-daemon .
```

Run the image in a container with `docker run`.

```
~/xray-daemon$ docker run \
      --attach STDOUT \
      -v ~/.aws/:/root/.aws/:ro \
      --net=host \
      -e AWS_REGION=us-east-2 \
      --name xray-daemon \
      -p 2000:2000/udp \
      xray-daemon -o
```

This command uses the following options:

- `--attach STDOUT` – View output from the daemon in the terminal.
- `-v ~/.aws/:/root/.aws/:ro` – Give the container read-only access to the `.aws` directory to let it read your AWS SDK credentials.
- `AWS_REGION=`*`us-east-2`* – Set the `AWS_REGION` environment variable to tell the daemon which region to use.
- `--net=host` – Attach the container to the `host` network. Containers on the host network can communicate with each other without publishing ports.
- `-p 2000:2000/udp` – Map UDP port 2000 on your machine to the same port on the container. This is not required for containers on the same network to communicate, but it does let you send segments to the daemon from the command line or from an application not running in Docker.
- `--name xray-daemon` – Name the container `xray-daemon` instead of generating a random name.
- `-o` (after the image name) – Append the `-o` option to the entry point that runs the daemon within the container. This option tells the daemon to run in local mode to prevent it from trying to read Amazon EC2 instance metadata.

To stop the daemon, use `docker stop`. If you make changes to the `Dockerfile` and build a new image, you need to delete the existing container before you can create another one with the same name. Use `docker rm` to delete the container.

```
$ docker stop xray-daemon
$ docker rm xray-daemon
```

Running the X-Ray daemon on Windows

You can run the daemon executable from the command line. Use the `-o` option to run in local mode, and `-n` to set the region.

```
> .\xray_windows.exe -o -n us-east-2
```

Use a PowerShell script to create and run a service for the daemon.

Example PowerShell script - Windows

```
if ( Get-Service "AWSXRayDaemon" -ErrorAction SilentlyContinue ){
    sc.exe stop AWSXRayDaemon
    sc.exe delete AWSXRayDaemon
}
if ( Get-Item -path aws-xray-daemon -ErrorAction SilentlyContinue ) {
    Remove-Item -Recurse -Force aws-xray-daemon
}

$currentLocation = Get-Location
$zipFileName = "aws-xray-daemon-windows-service-3.x.zip"
$zipPath = "$currentLocation\$zipFileName"
$destPath = "$currentLocation\aws-xray-daemon"
$daemonPath = "$destPath\xray.exe"
$daemonLogPath = "C:\inetpub\wwwroot\xray-daemon.log"
$url = "https://s3.dualstack.us-west-2.amazonaws.com/aws-xray-assets.us-west-2/xray-
daemon/aws-xray-daemon-windows-service-3.x.zip"

Invoke-WebRequest -Uri $url -OutFile $zipPath
Add-Type -Assembly "System.IO.Compression.Filesystem"
[io.compression.zipfile]::ExtractToDirectory($zipPath, $destPath)

sc.exe create AWSXRayDaemon binPath= "$daemonPath -f $daemonLogPath"
sc.exe start AWSXRayDaemon
```

Running the X-Ray daemon on OS X

You can run the daemon executable from the command line. Use the `-o` option to run in local mode, and `-n` to set the region.

```
~/xray-daemon$ ./xray_mac -o -n us-east-2
```

To run the daemon in the background, use &.

```
~/xray-daemon$ ./xray_mac -o -n us-east-2 &
```

Use nohup to prevent the daemon from terminating when the terminal is closed.

```
~/xray-daemon$ nohup ./xray_mac &
```

Running the X-Ray daemon on AWS Elastic Beanstalk

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

To relay trace data from your application to AWS X-Ray, you can run the X-Ray daemon on your Elastic Beanstalk environment's Amazon EC2 instances. For a list of supported platforms, see Configuring AWS X-Ray Debugging in the *AWS Elastic Beanstalk Developer Guide*.

> **Note**
>
> The daemon uses your environment's instance profile for permissions. For instructions about adding permissions to the Elastic Beanstalk instance profile, see Giving the daemon permission to send data to X-Ray.

Elastic Beanstalk platforms provide a configuration option that you can set to run the daemon automatically. You can enable the daemon in a configuration file in your source code or by choosing an option in the Elastic Beanstalk console. When you enable the configuration option, the daemon is installed on the instance and runs as a service.

The version included on Elastic Beanstalk platforms might not be the latest version. See the Supported Platforms topic to find out the version of the daemon that is available for your platform configuration.

Elastic Beanstalk does not provide the X-Ray daemon on the Multicontainer Docker (Amazon ECS) platform.

Using the Elastic Beanstalk X-Ray integration to run the X-Ray daemon

Use the console to turn on X-Ray integration, or configure it in your application source code with a configuration file.

To enable the X-Ray daemon in the Elastic Beanstalk console

1. Open the Elastic Beanstalk console.
2. Navigate to the management console for your environment.
3. Choose **Configuration**.
4. Choose **Software Settings**.
5. For **X-Ray daemon**, choose **Enabled**.
6. Choose **Apply**.

You can include a configuration file in your source code to make your configuration portable between environments.

Example .ebextensions/xray-daemon.config

```
option_settings:
  aws:elasticbeanstalk:xray:
    XRayEnabled: true
```

Elastic Beanstalk passes a configuration file to the daemon and outputs logs to a standard location.

On Windows Server Platforms

- **Configuration file** – `C:\Program Files\Amazon\XRay\cfg.yaml`
- **Logs** – `c:\Program Files\Amazon\XRay\logs\xray-service.log`

On Linux Platforms

- **Configuration file** – `/etc/amazon/xray/cfg.yaml`
- **Logs** – `/var/log/xray/xray.log`

Elastic Beanstalk provides tools for pulling instance logs from the AWS Management Console or command line. You can tell Elastic Beanstalk to include the X-Ray daemon logs by adding a task with a configuration file.

Example .ebextensions/xray-logs.config - Linux

```
files:
  "/opt/elasticbeanstalk/tasks/taillogs.d/xray-daemon.conf" :
    mode: "000644"
    owner: root
    group: root
    content: |
      /var/log/xray/xray.log
```

Example .ebextensions/xray-logs.config - Windows server

```
files:
  "c:/Program Files/Amazon/ElasticBeanstalk/config/taillogs.d/xray-daemon.conf" :
    mode: "000644"
    owner: root
    group: root
    content: |
      c:\Progam Files\Amazon\XRay\logs\xray-service.log
```

See Viewing Logs from Your Elastic Beanstalk Environment's Amazon EC2 Instances in the *AWS Elastic Beanstalk Developer Guide* for more information.

Downloading and running the X-Ray daemon manually (advanced)

If the X-Ray daemon isn't available for your platform configuration, you can download it from Amazon S3 and run it with a configuration file.

Use an Elastic Beanstalk configuration file to download and run the daemon.

Example .ebextensions/xray.config - Linux

```
commands:
  01-stop-tracing:
    command: yum remove -y xray
    ignoreErrors: true
  02-copy-tracing:
    command: curl https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/xray-
daemon/aws-xray-daemon-3.x.rpm -o /home/ec2-user/xray.rpm
  03-start-tracing:
    command: yum install -y /home/ec2-user/xray.rpm

files:
  "/opt/elasticbeanstalk/tasks/taillogs.d/xray-daemon.conf" :
    mode: "000644"
    owner: root
    group: root
    content: |
      /var/log/xray/xray.log
  "/etc/amazon/xray/cfg.yaml" :
    mode: "000644"
    owner: root
    group: root
    content: |
      Logging:
        LogLevel: "debug"
      Version: 2
```

Example .ebextensions/xray.config - Windows server

```
container_commands:
  01-execute-config-script:
    command: Powershell.exe -ExecutionPolicy Bypass -File c:\\temp\\installDaemon.ps1
    waitAfterCompletion: 0

files:
  "c:/temp/installDaemon.ps1":
    content: |
      if ( Get-Service "AWSXRayDaemon" -ErrorAction SilentlyContinue ) {
          sc.exe stop AWSXRayDaemon
          sc.exe delete AWSXRayDaemon
      }

      $targetLocation = "C:\Program Files\Amazon\XRay"
      if ((Test-Path $targetLocation) -eq 0) {
          mkdir $targetLocation
      }

      $zipFileName = "aws-xray-daemon-windows-service-3.x.zip"
      $zipPath = "$targetLocation\$zipFileName"
      $destPath = "$targetLocation\aws-xray-daemon"
      if ((Test-Path $destPath) -eq 1) {
          Remove-Item -Recurse -Force $destPath
      }

      $daemonPath = "$destPath\xray.exe"
      $daemonLogPath = "$targetLocation\xray-daemon.log"
      $url = "https://s3.dualstack.us-west-2.amazonaws.com/aws-xray-assets.us-west-2/
xray-daemon/aws-xray-daemon-windows-service-3.x.zip"

      Invoke-WebRequest -Uri $url -OutFile $zipPath
      Add-Type -Assembly "System.IO.Compression.Filesystem"
      [io.compression.zipfile]::ExtractToDirectory($zipPath, $destPath)

      New-Service -Name "AWSXRayDaemon" -StartupType Automatic -BinaryPathName
 "`"$daemonPath`" -f `"$daemonLogPath`""
      sc.exe start AWSXRayDaemon
    encoding: plain
  "c:/Program Files/Amazon/ElasticBeanstalk/config/taillogs.d/xray-daemon.conf" :
    mode: "000644"
    owner: root
    group: root
    content: |
      C:\Program Files\Amazon\XRay\xray-daemon.log
```

These examples also add the daemon's log file to the Elastic Beanstalk tail logs task, so that it's included when you request logs with the console or Elastic Beanstalk Command Line Interface (EB CLI).

Running the X-Ray daemon on Amazon EC2

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can run the X-Ray daemon on the following operating systems on Amazon EC2:

- Amazon Linux
- Ubuntu
- Windows Server (2012 R2 and newer)

Use an instance profile to grant the daemon permission to upload trace data to X-Ray. For more information, see Giving the daemon permission to send data to X-Ray.

Use a user data script to run the daemon automatically when you launch the instance.

Example User data script - Linux

```
#!/bin/bash
curl https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/xray-daemon/aws-xray-daemon-3.x.rpm -o /home/ec2-user/xray.rpm
yum install -y /home/ec2-user/xray.rpm
```

Example User data script - Windows server

```
<powershell>
if ( Get-Service "AWSXRayDaemon" -ErrorAction SilentlyContinue ) {
    sc.exe stop AWSXRayDaemon
    sc.exe delete AWSXRayDaemon
}

$targetLocation = "C:\Program Files\Amazon\XRay"
if ((Test-Path $targetLocation) -eq 0) {
    mkdir $targetLocation
}

$zipFileName = "aws-xray-daemon-windows-service-3.x.zip"
$zipPath = "$targetLocation\$zipFileName"
$destPath = "$targetLocation\aws-xray-daemon"
if ((Test-Path $destPath) -eq 1) {
    Remove-Item -Recurse -Force $destPath
}

$daemonPath = "$destPath\xray.exe"
$daemonLogPath = "$targetLocation\xray-daemon.log"
$url = "https://s3.dualstack.us-west-2.amazonaws.com/aws-xray-assets.us-west-2/xray-daemon/aws-xray-daemon-windows-service-3.x.zip"
```

```
Invoke-WebRequest -Uri $url -OutFile $zipPath
Add-Type -Assembly "System.IO.Compression.Filesystem"
[io.compression.zipfile]::ExtractToDirectory($zipPath, $destPath)

New-Service -Name "AWSXRayDaemon" -StartupType Automatic -BinaryPathName
 "`"$daemonPath`" -f `"$daemonLogPath`""
sc.exe start AWSXRayDaemon
</powershell>
```

Running the X-Ray daemon on Amazon ECS

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

In Amazon ECS, create a Docker image that runs the X-Ray daemon, upload it to a Docker image repository, and then deploy it to your Amazon ECS cluster. You can use port mappings and network mode settings in your task definition file to allow your application to communicate with the daemon container.

Using the official Docker image

X-Ray provides a Docker container image on Amazon ECR that you can deploy alongside your application. See downloading the daemon for more information.

Example Task definition

```
    {
      "name": "xray-daemon",
      "image": "amazon/aws-xray-daemon",
      "cpu": 32,
      "memoryReservation": 256,
      "portMappings" : [
          {
              "hostPort": 0,
              "containerPort": 2000,
              "protocol": "udp"
          }
       ]
    }
```

Create and build a Docker image

For custom configuration, you may need to define your own Docker image.

Add managed policies to your task role to grant the daemon permission to upload trace data to X-Ray. For more information, see Giving the daemon permission to send data to X-Ray.

Use one of the following Dockerfiles to create an image that runs the daemon.

Example Dockerfile – Amazon Linux

```
FROM amazonlinux
RUN yum install -y unzip
RUN curl -o daemon.zip https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/
xray-daemon/aws-xray-daemon-linux-3.x.zip
RUN unzip daemon.zip && cp xray /usr/bin/xray
ENTRYPOINT ["/usr/bin/xray", "-t", "0.0.0.0:2000", "-b", "0.0.0.0:2000"]
EXPOSE 2000/udp
EXPOSE 2000/tcp
```

Note

Flags -t and -b are required to specify a binding address to listen to the loopback of a multi-container environment.

Example Dockerfile – Ubuntu

For Debian derivatives, you also need to install certificate authority (CA) certificates to avoid issues when downloading the installer.

```
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y --force-yes --no-install-recommends apt-
transport-https curl ca-certificates wget && apt-get clean && apt-get autoremove && rm
 -rf /var/lib/apt/lists/*
RUN wget https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/xray-daemon/aws-
xray-daemon-3.x.deb
RUN dpkg -i aws-xray-daemon-3.x.deb
ENTRYPOINT ["/usr/bin/xray", "--bind=0.0.0.0:2000", "--bind-tcp=0.0.0.0:2000"]
EXPOSE 2000/udp
EXPOSE 2000/tcp
```

In your task definition, the configuration depends on the networking mode that you use. Bridge networking is the default and can be used in your default VPC. In a bridge network, set the AWS_XRAY_DAEMON_ADDRESS environment variable to tell the X-Ray SDK which container-port to reference and set the host port. For example, you could publish UDP port 2000, and create a link from your application container to the daemon container.

Example Task definition

```

    {
      "name": "xray-daemon",
      "image": "123456789012.dkr.ecr.us-east-2.amazonaws.com/xray-daemon",
      "cpu": 32,
      "memoryReservation": 256,
      "portMappings" : [
          {
              "hostPort": 0,
              "containerPort": 2000,
              "protocol": "udp"
          }
       ]
    },
    {
      "name": "scorekeep-api",
      "image": "123456789012.dkr.ecr.us-east-2.amazonaws.com/scorekeep-api",
      "cpu": 192,
      "memoryReservation": 512,
      "environment": [
          { "name" : "AWS_REGION", "value" : "us-east-2" },
          { "name" : "NOTIFICATION_TOPIC", "value" : "arn:aws:sns:us-
east-2:123456789012:scorekeep-notifications" },
          { "name" : "AWS_XRAY_DAEMON_ADDRESS", "value" : "xray-daemon:2000" }
      ],
      "portMappings" : [
          {
              "hostPort": 5000,
              "containerPort": 5000
          }
      ],
      "links": [
        "xray-daemon"
      ]
    }
```

If you run your cluster in the private subnet of a VPC, you can use the awsvpc network mode to attach an elastic network interface (ENI) to your containers. This enables you to avoid using links. Omit the host port in the port mappings, the link, and the `AWS_XRAY_DAEMON_ADDRESS` environment variable.

Example VPC task definition

```
{
    "family": "scorekeep",
    "networkMode":"awsvpc",
    "containerDefinitions": [
        {
          "name": "xray-daemon",
          "image": "123456789012.dkr.ecr.us-east-2.amazonaws.com/xray-daemon",
          "cpu": 32,
          "memoryReservation": 256,
          "portMappings" : [
              {
                  "containerPort": 2000,
                  "protocol": "udp"
              }
          ]
        },
        {
            "name": "scorekeep-api",
            "image": "123456789012.dkr.ecr.us-east-2.amazonaws.com/scorekeep-api",
            "cpu": 192,
            "memoryReservation": 512,
            "environment": [
                { "name" : "AWS_REGION", "value" : "us-east-2" },
                { "name" : "NOTIFICATION_TOPIC", "value" : "arn:aws:sns:us-
east-2:123456789012:scorekeep-notifications" }
            ],
            "portMappings" : [
                {
                    "containerPort": 5000
                }
            ]
        }
    ]
}
```

Configure command line options in the Amazon ECS console

Command line options override any conflicting values in your image's config file. Command line options are typically used for local testing, but can also be used for convenience while setting environment variables, or to control the startup process.

By adding command line options, you are updating the Docker `CMD` that is passed to the container. For more information, see [the Docker run reference](https://docs.docker.com/engine/reference/run/).

To set a command line option

1. Open the Amazon ECS classic console at https://console.aws.amazon.com/ecs/.
2. From the navigation bar, choose the region that contains your task definition.
3. In the navigation pane, choose **Task Definitions**.
4. On the **Task Definitions** page, select the box to the left of the task definition to revise and choose **Create new revision**.
5. On the **Create new revision of Task Definition** page, select the container.
6. In the **ENVIRONMENT** section, add your comma-separated list of command line options to the **Command** field.
7. Choose **Update**.
8. Verify the information and choose **Create**.

The following example shows how to write a comma-separated command line option for the `RoleARN` option. The `RoleARN` option assumes the specified IAM role to upload segments to a different account.

Example

```
--role-arn, arn:aws:iam::123456789012:role/xray-cross-account
```

To learn more about the available command line options in X-Ray, see Configuring the AWS X-Ray Daemon.

Integrating AWS X-Ray with other AWS services

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

Many AWS services provide varying levels of X-Ray integration, including sampling and adding headers to incoming requests, running the X-Ray daemon, and automatically sending trace data to X-Ray. Integration with X-Ray can include the following:

- *Active instrumentation* – Samples and instruments incoming requests
- *Passive instrumentation* – Instruments requests that have been sampled by another service
- *Request tracing* – Adds a tracing header to all incoming requests and propagates it downstream
- *Tooling* – Runs the X-Ray daemon to receive segments from the X-Ray SDK

Note

The X-Ray SDKs include plugins for additional integration with AWS services. For example, you can use the X-Ray SDK for Java Elastic Beanstalk plugin to add information about the Elastic Beanstalk environment that runs your application, including the environment name and ID.

Here are some examples of AWS services that are integrated with X-Ray:

- AWS Distro for OpenTelemetry (ADOT) – With ADOT, engineers can instrument their applications once and send correlated metrics and traces to multiple AWS monitoring solutions including Amazon CloudWatch, AWS X-Ray, Amazon OpenSearch Service, and Amazon Managed Service for Prometheus.

- AWS Lambda – Active and passive instrumentation of incoming requests on all runtimes. AWS Lambda adds two nodes to your trace map, one for the AWS Lambda service, and one for the function. When you enable instrumentation, AWS Lambda also runs the X-Ray daemon on Java and Node.js runtimes for use with the X-Ray SDK.
- Amazon API Gateway – Active and passive instrumentation. API Gateway uses sampling rules to determine which requests to record, and adds a node for the gateway stage to your service map.
- AWS Elastic Beanstalk – Tooling. Elastic Beanstalk includes the X-Ray daemon on the following platforms:
  - *Java SE* – 2.3.0 and later configurations
  - *Tomcat* – 2.4.0 and later configurations
  - *Node.js* – 3.2.0 and later configurations
  - *Windows Server* – All configurations other than Windows Server Core that have been released after December 9th, 2016

  You can use the Elastic Beanstalk console to tell Elastic Beanstalk to run the daemon on these platforms, or use the `XRayEnabled` option in the `aws:elasticbeanstalk:xray` namespace.
- Elastic Load Balancing – Request tracing on Application Load Balancers. The Application Load Balancer adds the trace ID to the request header before sending it to a target group.
- Amazon EventBridge – Passive instrumentation. If a service that publishes events to EventBridge is instrumented with the X-Ray SDK, event targets will receive the tracing header and can continue to propagate the original trace ID.
- Amazon Simple Notification Service – Passive instrumentation. If an Amazon SNS publisher traces its client with the X-Ray SDK, subscribers can retrieve the tracing header and continue to propagate the original trace from the publisher with the same trace ID.
- Amazon Simple Queue Service – Passive instrumentation. If a service traces requests by using the X-Ray SDK, Amazon SQS can send the tracing header and continue to propagate the original trace from the sender to the consumer with a consistent trace ID.
- Amazon Bedrock AgentCore – AgentCore supports distributed tracing through X-Ray integration, allowing you to track requests as they flow through your agent applications. When you enable observability for your AgentCore resources, you can propagate trace context across service boundaries and gain visibility into the performance of your AI agents and tools.

Choose from the following topics to explore the full set of integrated AWS services.

Topics

- Amazon Bedrock AgentCore and AWS X-Ray
- Amazon Elastic Compute Cloud and AWS X-Ray
- Amazon SNS and AWS X-Ray
- Amazon SQS and AWS X-Ray
- Amazon S3 and AWS X-Ray
- AWS Distro for OpenTelemetry and AWS X-Ray
- Tracking X-Ray encryption configuration changes with AWS Config
- AWS AppSync and AWS X-Ray
- Amazon API Gateway active tracing support for AWS X-Ray
- Amazon EC2 and AWS App Mesh
- AWS App Runner and X-Ray
- Logging X-Ray API calls with AWS CloudTrail
- CloudWatch integration with X-Ray
- AWS Elastic Beanstalk and AWS X-Ray
- Elastic Load Balancing and AWS X-Ray
- Amazon EventBridge and AWS X-Ray
- AWS Lambda and AWS X-Ray
- AWS Step Functions and AWS X-Ray

Amazon Bedrock AgentCore and AWS X-Ray

Amazon Bedrock AgentCore integrates with AWS X-Ray to provide distributed tracing capabilities for your AI agents and tools. This integration allows you to track requests as they flow through your agent applications, helping you identify performance bottlenecks and troubleshoot issues.

AgentCore supports distributed tracing through X-Ray integration, allowing you to monitor the performance of your AI agents and tools. When you enable observability for your AgentCore resources, you can propagate trace context across service boundaries and gain visibility into how your agents interact with other AWS services. For more information, see Amazon Bedrock AgentCore.

AgentCore supports the following X-Ray features:

- Propagation of trace context to downstream services
- Custom instrumentation using the AWS Distro for OpenTelemetry (ADOT) SDK

Setting up X-Ray with AgentCore

To use X-Ray with AgentCore, you need to enable CloudWatch Transaction Search in your AWS account. This is a one-time setup that allows AgentCore to send trace data to X-Ray. For more information, see Enable transaction search .

For more information about setting up observability for AgentCore, see Add observability to your Amazon Bedrock AgentCore agent or tool .

Using trace headers with AgentCore

AgentCore supports the X-Ray trace header format for distributed tracing. You can include the `X-Amzn-Trace-Id` header in your requests to AgentCore to maintain trace context across service boundaries.

Amazon Elastic Compute Cloud and AWS X-Ray

You can install and run the X-Ray daemon on an Amazon EC2 instance with a user data script. See Running the X-Ray daemon on Amazon EC2 for instructions.

Use an instance profile to grant the daemon permission to upload trace data to X-Ray. For more information, see Giving the daemon permission to send data to X-Ray.

Amazon SNS and AWS X-Ray

You can use AWS X-Ray with Amazon Simple Notification Service (Amazon SNS) to trace and analyze requests as they travel through your SNS topics to your SNS-supported subscription services. Use X-Ray tracing with Amazon SNS to analyze latencies in your messages and their back-end services, such as how long a request spends in a topic, and how long it takes to deliver the message to each of the topic’s subscriptions. Amazon SNS supports X-Ray tracing for both standard and FIFO topics.

If you publish to an Amazon SNS topic from a service that’s already instrumented with X-Ray, Amazon SNS passes the trace context from publisher to subscribers. In addition, you can turn on active tracing to send segment data about your Amazon SNS subscriptions to X-Ray for messages published from an instrumented SNS client. Turn on active tracing for an Amazon SNS topic by using the Amazon SNS console, or by using the Amazon SNS API or CLI. See Instrumenting your application for more information about instrumenting your SNS clients.

Configure Amazon SNS active tracing

You can use the Amazon SNS console or the AWS CLI or SDK to configure Amazon SNS active tracing.

When you use the Amazon SNS console, Amazon SNS attempts to create the necessary permissions for SNS to call X-Ray. The attempt can be rejected if you don't have sufficient permissions to modify X-Ray resource policies. For more information about these permissions, see Identity and access management in Amazon SNS and Example cases for Amazon SNS access control in the Amazon Simple Notification Service Developer Guide. For more information about turning on active tracing using the Amazon SNS console, see Enabling active tracing on an Amazon SNS topic in the Amazon Simple Notification Service Developer Guide.

When using the AWS CLI or SDK to turn on active tracing, you must manually configure the permissions using resource-based policies. Use `PutResourcePolicy` to configure X-Ray with the necessary resource-based policy to allow Amazon SNS to send traces to X-Ray.

Example Example X-Ray resource-based policy for Amazon SNS active tracing

This example policy document specifies the permissions that Amazon SNS needs to send trace data to X-Ray:

```
{
    Version: "2012-10-17",
    Statement: [
      {
        Sid: "SNSAccess",
        Effect: Allow,
        Principal: {
          Service: "sns.amazonaws.com",
        },
        Action: [
          "xray:PutTraceSegments",
          "xray:GetSamplingRules",
          "xray:GetSamplingTargets"
        ],
        Resource: "*",
        Condition: {
          StringEquals: {
            "aws:SourceAccount": "account-id"
          },
          StringLike: {
            "aws:SourceArn": "arn:partition:sns:region:account-id:topic-name"
          }
        }
      }
    ]
  }
```

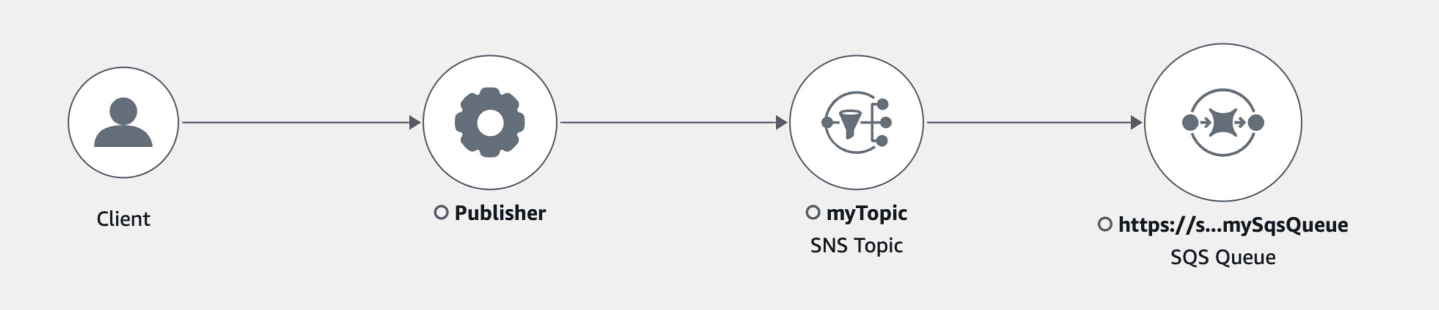
Use the CLI to create a resource-based policy that gives Amazon SNS permissions to send trace data to X-Ray:

```
aws xray put-resource-policy --policy-name MyResourcePolicy --policy-document
 '{ "Version": "2012-10-17", "Statement": [ { "Sid": "SNSAccess", "Effect": "Allow",
 "Principal": { "Service": "sns.amazonaws.com" }, "Action": [ "xray:PutTraceSegments",
 "xray:GetSamplingRules", "xray:GetSamplingTargets" ], "Resource": "*",
 "Condition": { "StringEquals": { "aws:SourceAccount": "account-id" }, "StringLike":
 { "aws:SourceArn": "arn:partition:sns:region:account-id:topic-name" } } } ] }'
```

To use these examples, replace *`partition`*, *`region`*, *`account-id`*, and *`topic-name`* with your specific AWS partition, region, account ID, and Amazon SNS topic name. To give all Amazon SNS topics permission to send trace data to X-Ray, replace the topic name with *.

View Amazon SNS publisher and subscriber traces in the X-Ray console

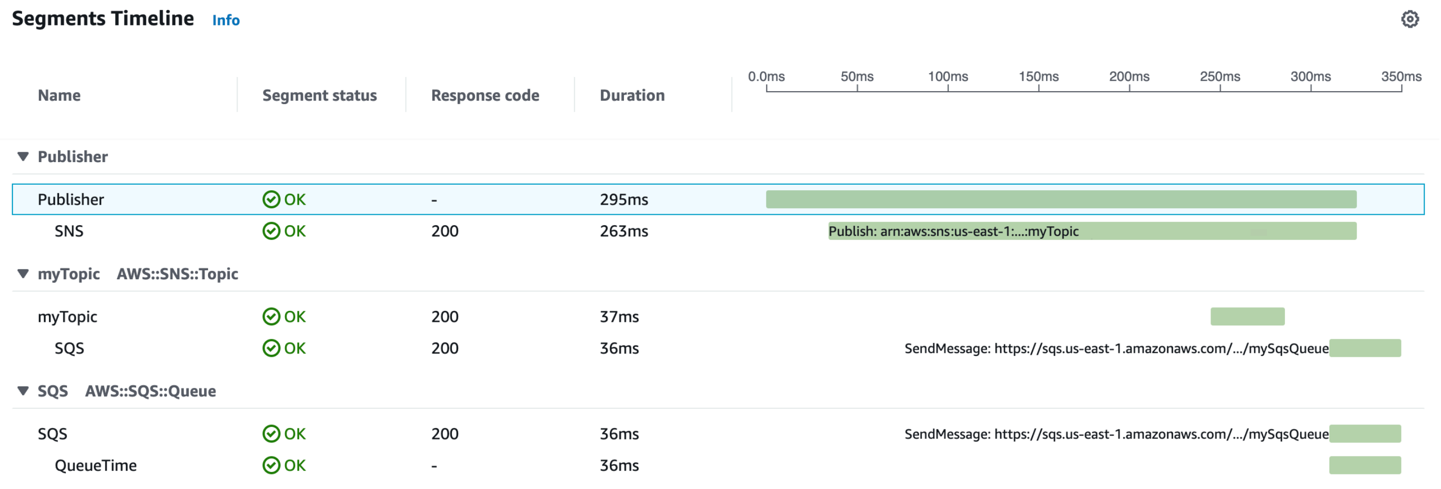
Use the X-Ray console to view a trace map and trace details that display a connected view of Amazon SNS publishers and subscribers. When Amazon SNS active tracing is turned on for a topic, the X-Ray trace map and trace details map displays connected nodes for Amazon SNS publishers, the Amazon SNS topic, and downstream subscribers:



After choosing a trace that spans an Amazon SNS publisher and subscriber, the X-Ray trace details page displays a trace details map and segment timeline.

Example Example timeline with Amazon SNS publisher and subscriber

This example shows a timeline that includes an Amazon SNS publisher that sends a message to an Amazon SNS topic, which is processed by an Amazon SQS subscriber.



Segments Timeline Info

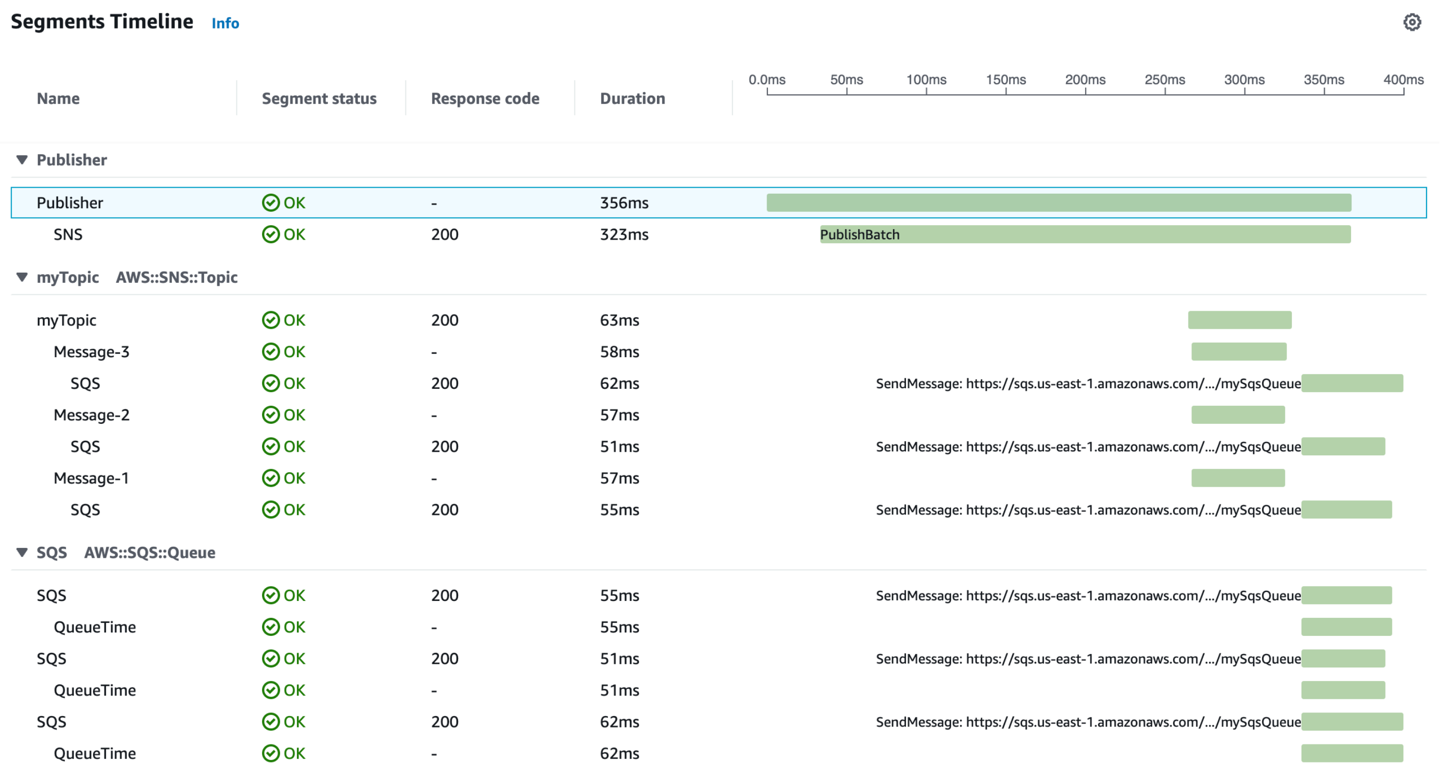
Name	Segment status	Response code	Duration	
▼ **Publisher**				
Publisher	✓ OK	-	295ms	
SNS	✓ OK	200	263ms	Publish: arn:aws:sns:us-east-1:...:myTopic
▼ **myTopic** **AWS::SNS::Topic**				
myTopic	✓ OK	200	37ms	
SQS	✓ OK	200	36ms	SendMessage: https://sqs.us-east-1.amazonaws.com/.../mySqsQueue
▼ **SQS** **AWS::SQS::Queue**				
SQS	✓ OK	200	36ms	SendMessage: https://sqs.us-east-1.amazonaws.com/.../mySqsQueue
QueueTime	✓ OK	-	36ms	

The example timeline above provides details about the Amazon SNS message flow:

- The **SNS** segment represents the round-trip duration of the `Publish` API call from the client.
- The **myTopic** segment represents the latency of the Amazon SNS response to the publish request.
- The **SQS** subsegment represents the round-trip time it takes Amazon SNS to publish the message to an Amazon SQS queue.
- The time between the **myTopic** segment and the **SQS** subsegment represents the time that the message spends in the Amazon SNS system.

Example Example timeline with batched Amazon SNS messages

If multiple Amazon SNS messages are batched within a single trace, the segment timeline displays segments that represent each message that's processed.



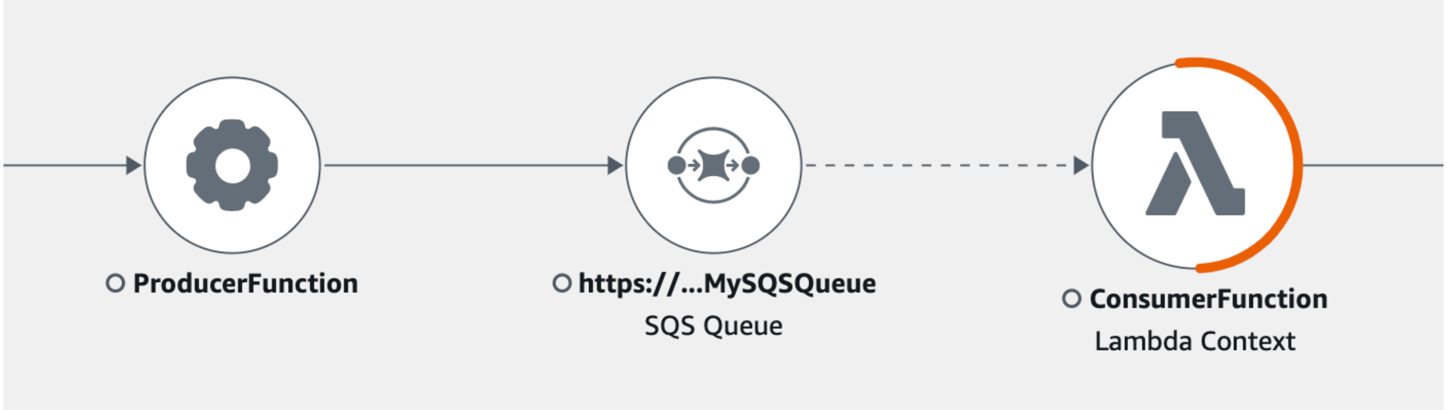
Segments Timeline Info

Name	Segment status	Response code	Duration	
▼ Publisher				
Publisher	✓ OK	-	356ms	
SNS	✓ OK	200	323ms	PublishBatch
▼ myTopic AWS::SNS::Topic				
myTopic	✓ OK	200	63ms	
Message-3	✓ OK	-	58ms	
SQS	✓ OK	200	62ms	SendMessage: https://sqs.us-east-1.amazonaws.com/.../mySqsQueue
Message-2	✓ OK	-	57ms	
SQS	✓ OK	200	51ms	SendMessage: https://sqs.us-east-1.amazonaws.com/.../mySqsQueue
Message-1	✓ OK	-	57ms	
SQS	✓ OK	200	55ms	SendMessage: https://sqs.us-east-1.amazonaws.com/.../mySqsQueue
▼ SQS AWS::SQS::Queue				
SQS	✓ OK	200	55ms	SendMessage: https://sqs.us-east-1.amazonaws.com/.../mySqsQueue
QueueTime	✓ OK	-	55ms	
SQS	✓ OK	200	51ms	SendMessage: https://sqs.us-east-1.amazonaws.com/.../mySqsQueue
QueueTime	✓ OK	-	51ms	
SQS	✓ OK	200	62ms	SendMessage: https://sqs.us-east-1.amazonaws.com/.../mySqsQueue
QueueTime	✓ OK	-	62ms	

Amazon SQS and AWS X-Ray

AWS X-Ray integrates with Amazon Simple Queue Service (Amazon SQS) to trace messages that are passed through an Amazon SQS queue. If a service traces requests by using the X-Ray SDK, Amazon SQS can send the tracing header and continue to propagate the original trace from the sender to the consumer with a consistent trace ID. Trace continuity enables users to track, analyze, and debug throughout downstream services.

AWS X-Ray supports tracing event-driven applications using Amazon SQS and AWS Lambda. Use the CloudWatch console to see a connected view of each request as it's queued with Amazon SQS and processed by a downstream Lambda function. Traces from upstream message producers are automatically linked to traces from downstream Lambda consumer nodes, creating an end-to-end view of the application. For more information, see tracing event-driven applications.



Amazon SQS supports the following tracing header instrumentation:

- **Default HTTP Header** – The X-Ray SDK automatically populates the trace header as an HTTP header when you call Amazon SQS through the AWS SDK. The default trace header is carried by `X-Amzn-Trace-Id` and corresponds to all messages included in a `SendMessage` or `SendMessageBatch` request. To learn more about the default HTTP header, see Tracing header.
- **`AWSTraceHeader` System Attribute** – The `AWSTraceHeader` is a message system attribute reserved by Amazon SQS to carry the X-Ray trace header with messages in the queue. `AWSTraceHeader` is available for use even when auto-instrumentation through the X-Ray SDK is not, for example when building a tracing SDK for a new language. When both header instrumentations are set, the message system attribute overrides the HTTP trace header.

When running on Amazon EC2, Amazon SQS supports processing one message at a time. This applies when running on an on-premises host, and when using container services, such as AWS Fargate, Amazon ECS, or AWS App Mesh.

The trace header is excluded from both Amazon SQS message size and message attribute quotas. Enabling X-Ray tracing will not exceed your Amazon SQS quotas. To learn more about AWS quotas, see Amazon SQS Quotas.

Send the HTTP trace header

Sender components in Amazon SQS can send the trace header automatically through the `SendMessageBatch` or `SendMessage` call. When AWS SDK clients are instrumented, they can be automatically tracked through all languages supported through the X-Ray SDK. Traced AWS services and resources that you access within those services (for example, an Amazon S3 bucket or Amazon SQS queue), appear as downstream nodes on the trace map in the X-Ray console.

To learn how to trace AWS SDK calls with your preferred language, see the following topics in the supported SDKs:

- Go – Tracing AWS SDK calls with the X-Ray SDK for Go
- Java – Tracing AWS SDK calls with the X-Ray SDK for Java
- Node.js – Tracing AWS SDK calls with the X-Ray SDK for Node.js
- Python – Tracing AWS SDK calls with the X-Ray SDK for Python
- Ruby – Tracing AWS SDK calls with the X-Ray SDK for Ruby
- .NET – Tracing AWS SDK calls with the X-Ray SDK for .NET

Retrieve the trace header and recover trace context

If you are using a Lambda downstream consumer, trace context propagation is automatic. To continue context propagation with other Amazon SQS consumers, you must manually instrument the handoff to the receiver component.

There are three main steps to recovering the trace context:

- Receive the message from the queue for the `AWSTraceHeader` attribute by calling the `ReceiveMessage` API.
- Retrieve the trace header from the attribute.
- Recover the trace ID from the header. Optionally, add more metrics to the segment.

The following is an example implementation written with the X-Ray SDK for Java.

Example : Retrieve the trace header and recover trace context

```
// Receive the message from the queue, specifying the "AWSTraceHeader"
ReceiveMessageRequest receiveMessageRequest = new ReceiveMessageRequest()
        .withQueueUrl(QUEUE_URL)
        .withAttributeNames("AWSTraceHeader");
List<Message> messages = sqs.receiveMessage(receiveMessageRequest).getMessages();

if (!messages.isEmpty()) {
    Message message = messages.get(0);

    // Retrieve the trace header from the AWSTraceHeader message system attribute
    String traceHeaderStr = message.getAttributes().get("AWSTraceHeader");
    if (traceHeaderStr != null) {
        TraceHeader traceHeader = TraceHeader.fromString(traceHeaderStr);

        // Recover the trace context from the trace header
        Segment segment = AWSXRay.getCurrentSegment();
        segment.setTraceId(traceHeader.getRootTraceId());
        segment.setParentId(traceHeader.getParentId());

 segment.setSampled(traceHeader.getSampled().equals(TraceHeader.SampleDecision.SAMPLED));
    }
}
```

Amazon S3 and AWS X-Ray

AWS X-Ray integrates with Amazon S3 to trace upstream requests to update your application's S3 buckets. If a service traces requests by using the X-Ray SDK, Amazon S3 can send the tracing headers to downstream event subscribers such as AWS Lambda, Amazon SQS, and Amazon SNS. X-Ray enables trace messages for Amazon S3 event notifications.

You can use the X-Ray trace map to view the connections between Amazon S3 and other services that your application uses. You can also use the console to view metrics such as average latency and failure rates. For more information about the X-Ray console, see Use the X-Ray console.

Amazon S3 supports the *default http header* instrumentation. The X-Ray SDK automatically populates the trace header as an HTTP header when you call Amazon S3 through the AWS SDK. The default trace header is carried by `X-Amzn-Trace-Id`. To learn more about tracing headers, see Tracing header on the concept page. Amazon S3 trace context propagation supports the following subscribers: Lambda, SQS and SNS. Because SQS and SNS don't emit segment data themselves, they won't appear in your trace or trace map when triggered by S3, even though they will propagate the tracing header to downstream services.

Configure Amazon S3 event notifications

With the Amazon S3 notification feature, you receive notifications when certain events happen in your bucket. These notifications can then be propagated to the following destinations within your application:

- Amazon Simple Notification Service (Amazon SNS)
- Amazon Simple Queue Service (Amazon SQS)

- AWS Lambda

For a list of supported events, see Supported event types in the Amazon S3 developer guide.

Amazon SNS and Amazon SQS

To publish notifications to an SNS topic or an SQS queue, you must first grant Amazon S3 permissions. To grant these permissions, you attach an AWS Identity and Access Management (IAM) policy to the destination SNS topic or SQS queue. To learn more about the IAM policies required, see Granting permissions to publish messages to an SNS topic or an SQS queue.

For information about integrating SNS and SQS with X-Ray see, Amazon SNS and AWS X-Ray and Amazon SQS and AWS X-Ray.

AWS Lambda

When you use the Amazon S3 console to configure event notifications on an S3 bucket for a Lambda function, the console sets up the necessary permissions on the Lambda function so that Amazon S3 has permissions to invoke the function from the bucket. For more information, see How Do I Enable and Configure Event Notifications for an S3 Bucket? in the Amazon Simple Storage Service Console User Guide.

You can also grant Amazon S3 permissions from AWS Lambda to invoke your Lambda function. For more information, see Tutorial: Using AWS Lambda with Amazon S3 in the AWS Lambda Developer Guide.

For more information about integrating Lambda with X-Ray, see Instrumenting Java code in AWS Lambda.

AWS Distro for OpenTelemetry and AWS X-Ray

Use the AWS Distro for OpenTelemetry (ADOT) to collect and send metrics and traces to AWS X-Ray and other monitoring solutions, such as Amazon CloudWatch, Amazon OpenSearch Service, and Amazon Managed Service for Prometheus.

AWS Distro for OpenTelemetry

The AWS Distro for OpenTelemetry (ADOT) is an AWS distribution based on the Cloud Native Computing Foundation (CNCF) OpenTelemetry project. OpenTelemetry provides a single set of open source APIs, libraries, and agents to collect distributed traces and metrics. This toolkit is a distribution of upstream OpenTelemetry components including SDKs, auto-instrumentation agents, and collectors that are tested, optimized, secured, and supported by AWS.

With ADOT, engineers can instrument their applications once and send correlated metrics and traces to multiple AWS monitoring solutions including Amazon CloudWatch, AWS X-Ray, Amazon OpenSearch Service, and Amazon Managed Service for Prometheus.

ADOT is integrated with a growing number of AWS services to simplify sending traces and metrics to monitoring solutions such as X-Ray. Some examples of services integrated with ADOT include:

- *AWS Lambda* – AWS managed Lambda layers for ADOT provides a plug-and-play user experience by automatically instrumenting a Lambda function, packaging OpenTelemetry together with an out-of-the-box configuration for AWS Lambda and X-Ray in an easy to setup layer. Users can enable and disable OpenTelemetry for their Lambda function without changing code. For more information, see AWS Distro for OpenTelemetry Lambda
- *Amazon Elastic Container Service (ECS)* – Collect metrics and traces from Amazon ECS applications using the AWS Distro for OpenTelemetry Collector, to send to X-Ray and other monitoring solutions. For more information, see Collecting application trace data in the Amazon ECS developer guide.
- *AWS App Runner* – App Runner supports sending traces to X-Ray using the AWS Distro for OpenTelemetry (ADOT). Use ADOT SDKs to collect trace data for your containerized applications, and use X-Ray to analyze and gain insights into your instrumented application. For more information, see AWS App Runner and X-Ray.

For more information about the AWS Distro for OpenTelemetry, including integration with additional AWS services, see the AWS Distro for OpenTelemetry Documentation.

For more information about instrumenting your application with AWS Distro for OpenTelemetry and X-Ray, see Instrumenting your application with the AWS Distro for OpenTelemetry.

Tracking X-Ray encryption configuration changes with AWS Config

AWS X-Ray integrates with AWS Config to record configuration changes made to your X-Ray encryption resources. You can use AWS Config to inventory X-Ray encryption resources, audit the X-Ray configuration history, and send notifications based on resource changes.

AWS Config supports logging the following X-Ray encryption resource changes as events:

- **Configuration changes –** Changing or adding an encryption key, or reverting to the default X-Ray encryption setting.

Use the following instructions to learn how to create a basic connection between X-Ray and AWS Config.

Creating a Lambda function trigger

You must have the ARN of a custom AWS Lambda function before you can generate a custom AWS Config rule. Follow these instructions to create a basic function with Node.js that returns a compliant or non-compliant value back to AWS Config based on the state of the `XrayEncryptionConfig` resource.

To create a Lambda function with an AWS::XrayEncryptionConfig change trigger

1. Open the Lambda console. Choose **Create function**.
2. Choose **Blueprints**, and then filter the blueprints library for the **config-rule-change-triggered** blueprint. Either click the link in the blueprint's name or choose **Configure** to continue.
3. Define the following fields to configure the blueprint:
   - For **Name**, type a name.
   - For **Role**, choose **Create new role from template(s)**.
   - For **Role name**, type a name.
   - For **Policy templates**, choose **AWS Config Rules permissions**.
4. Choose **Create function** to create and display your function in the AWS Lambda console.
5. Edit your function code to replace `AWS::EC2::Instance` with `AWS::XrayEncryptionConfig`. You can also update the description field to reflect this change.

   Default Code

```
   if (configurationItem.resourceType !== 'AWS::EC2::Instance') {
       return 'NOT_APPLICABLE';
   } else if (ruleParameters.desiredInstanceType ===
configurationItem.configuration.instanceType) {
       return 'COMPLIANT';
   }
       return 'NON_COMPLIANT';
```

Updated Code

```
   if (configurationItem.resourceType !== 'AWS::XRay::EncryptionConfig') {
       return 'NOT_APPLICABLE';
   } else if (ruleParameters.desiredInstanceType ===
configurationItem.configuration.instanceType) {
       return 'COMPLIANT';
   }
       return 'NON_COMPLIANT';
```

6. Add the following to your execution role in IAM for access to X-Ray. These permissions allow read-only access to your X-Ray resources. Failure to provide access to the appropriate resources will result in an out of scope message from AWS Config when it evaluates the Lambda function associated with the rule.

```
   {
       "Sid": "Stmt1529350291539",
       "Action": [
           "xray:GetEncryptionConfig"
       ],
       "Effect": "Allow",
       "Resource": "*"
    }
```

Creating a custom AWS Config rule for x-ray

When the Lambda function is created, note the function's ARN, and go to the AWS Config console to create your custom rule.

To create an AWS Config rule for X-Ray

1. Open the **Rules** page of the AWS Config console.
2. Choose **Add rule**, and then choose **Add custom rule**.
3. In **AWS Lambda Function ARN**, insert the ARN associated with the Lambda function you want to use.
4. Choose the type of trigger to set:

- **Configuration changes** – AWS Config triggers the evaluation when any resource that matches the rule's scope changes in configuration. The evaluation runs after AWS Config sends a configuration item change notification.
- **Periodic** – AWS Config runs evaluations for the rule at a frequency that you choose (for example, every 24 hours).

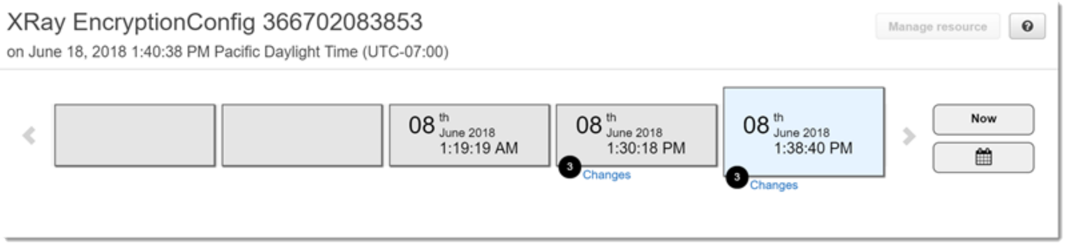
5. For **Resource type**, choose **EncryptionConfig** in the X-Ray section.
6. Choose **Save**.

The AWS Config console begins to evaluate the rule's compliance immediately. The evaluation can take several minutes to complete.

Now that this rule is compliant, AWS Config can begin to compile an audit history. AWS Config records resource changes in the form of a timeline. For each change in the timeline of events, AWS Config generates a table in a from/to format to show what changed in the JSON representation of the encryption key. The two field changes associated with EncryptionConfig are `Configuration.type` and `Configuration.keyID`.

Example results

Following is an example of an AWS Config timeline showing changes made at specific dates and times.

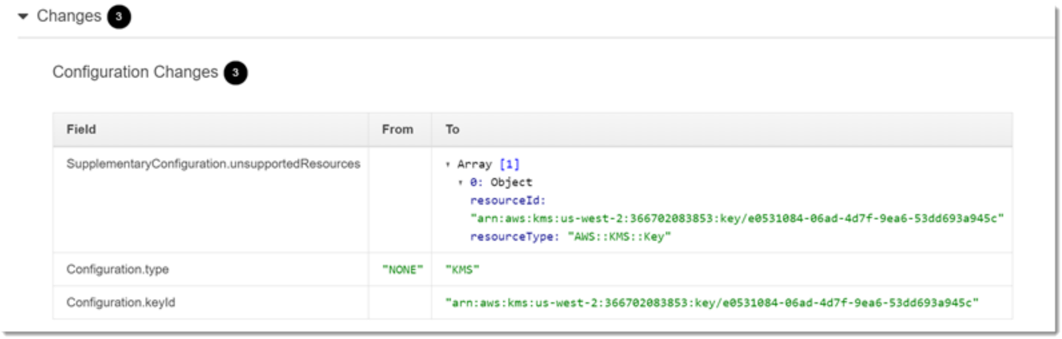


XRay EncryptionConfig 366702083853

on June 18, 2018 1:40:38 PM Pacific Daylight Time (UTC-07:00)

Manage resource
08 th June 2018 1:19:19 AM
08 th June 2018 1:30:18 PM
3 Changes
08 th June 2018 1:38:40 PM
3 Changes
Now

Following is an example of an AWS Config change entry. The from/to format illustrates what changed. This example shows that the default X-Ray encryption settings were changed to a defined encryption key.



▾ Changes 3

Configuration Changes 3

Field	From	To
SupplementaryConfiguration.unsupportedResources		▾ Array [1] ▾ 0: Object resourceId: "arn:aws:kms:us-west-2:366702083853:key/e0531084-06ad-4d7f-9ea6-53dd693a945c" resourceType: "AWS::KMS::Key"
Configuration.type	"NONE"	"KMS"
Configuration.keyId		"arn:aws:kms:us-west-2:366702083853:key/e0531084-06ad-4d7f-9ea6-53dd693a945c"

Amazon SNS notifications

To be notified of configuration changes, set AWS Config to publish Amazon SNS notifications. For more information, see Monitoring AWS Config Resource Changes by Email.

AWS AppSync and AWS X-Ray

You can enable and trace requests for AWS AppSync. For more information, see Tracing with AWS X-Ray for instructions.

When X-Ray tracing is enabled for an AWS AppSync API, an AWS Identity and Access Management service-linked role is automatically created in your account with the appropriate permissions. This allows AWS AppSync to send traces to X-Ray in a secure way.

Amazon API Gateway active tracing support for AWS X-Ray

You can use X-Ray to trace and analyze user requests as they travel through your Amazon API Gateway APIs to the underlying services. API Gateway supports X-Ray tracing for all API Gateway endpoint types: Regional, edge-optimized, and private. You can use X-Ray with Amazon API Gateway in all AWS Regions where X-Ray is available. For more information, see Trace API Gateway API Execution with AWS X-Ray in the Amazon API Gateway Developer Guide.

> **Note**
>
> X-Ray only supports tracing for REST APIs through API Gateway.

Amazon API Gateway provides active tracing support for AWS X-Ray. Enable active tracing on your API stages to sample incoming requests and send traces to X-Ray.

To enable active tracing on an API stage

1. Open the API Gateway console at https://console.aws.amazon.com/apigateway/.
2. Choose an API.
3. Choose a stage.
4. On the **Logs/Tracing** tab, choose **Enable X-Ray Tracing** and then choose **Save Changes**.
5. Choose **Resources** in the left side navigation panel.
6. To redeploy the API with the new settings, choose the **Actions** dropdown, and then choose **Deploy API**.

API Gateway uses sampling rules that you define in the X-Ray console to determine which requests to record. You can create rules that only apply to APIs, or that apply only to requests that contain certain headers. API Gateway records headers in attributes on the segment, along with details about the stage and request. For more information, see Configuring sampling rules.

> **Note**
>
> When tracing REST APIs with API Gateway HTTP integration, each segment's service name is set to the request URL path from API Gateway to your HTTP integration endpoint, resulting in a service node on the X-Ray trace map for each unique URL path. A large number of URL paths may cause the trace map to exceed the limit of 10,000 nodes, resulting in an error.
> To minimize the number of service nodes created by API Gateway, consider passing parameters within the URL query string or in the request body via POST. Either approach will ensure parameters are not part of the URL path, which may result in fewer distinct URL paths and service nodes.

For all incoming requests, API Gateway adds a tracing header to incoming HTTP requests that don't already have one.

```
X-Amzn-Trace-Id: Root=1-5759e988-bd862e3fe1be46a994272793
```

X-Ray trace ID format

An X-Ray `trace_id` consists of three numbers separated by hyphens. For example, `1-58406520-a006649127e371903a2de979`. This includes:

- The version number, which is 1.
- The time of the original request in Unix epoch time using **8 hexadecimal digits**.

  For example, 10:00AM December 1st, 2016 PST in epoch time is `1480615200` seconds or `58406520` in hexadecimal digits.
- A globally unique 96-bit identifier for the trace in **24 hexadecimal digits**.

If active tracing is disabled, the stage still records a segment if the request comes from a service that sampled the request and started a trace. For example, an instrumented web application can call an API Gateway API with an HTTP client. When you instrument an HTTP client with the X-Ray SDK, it adds a tracing header to the outgoing request that contains the sampling decision. API Gateway reads the tracing header and creates a segment for sampled requests.

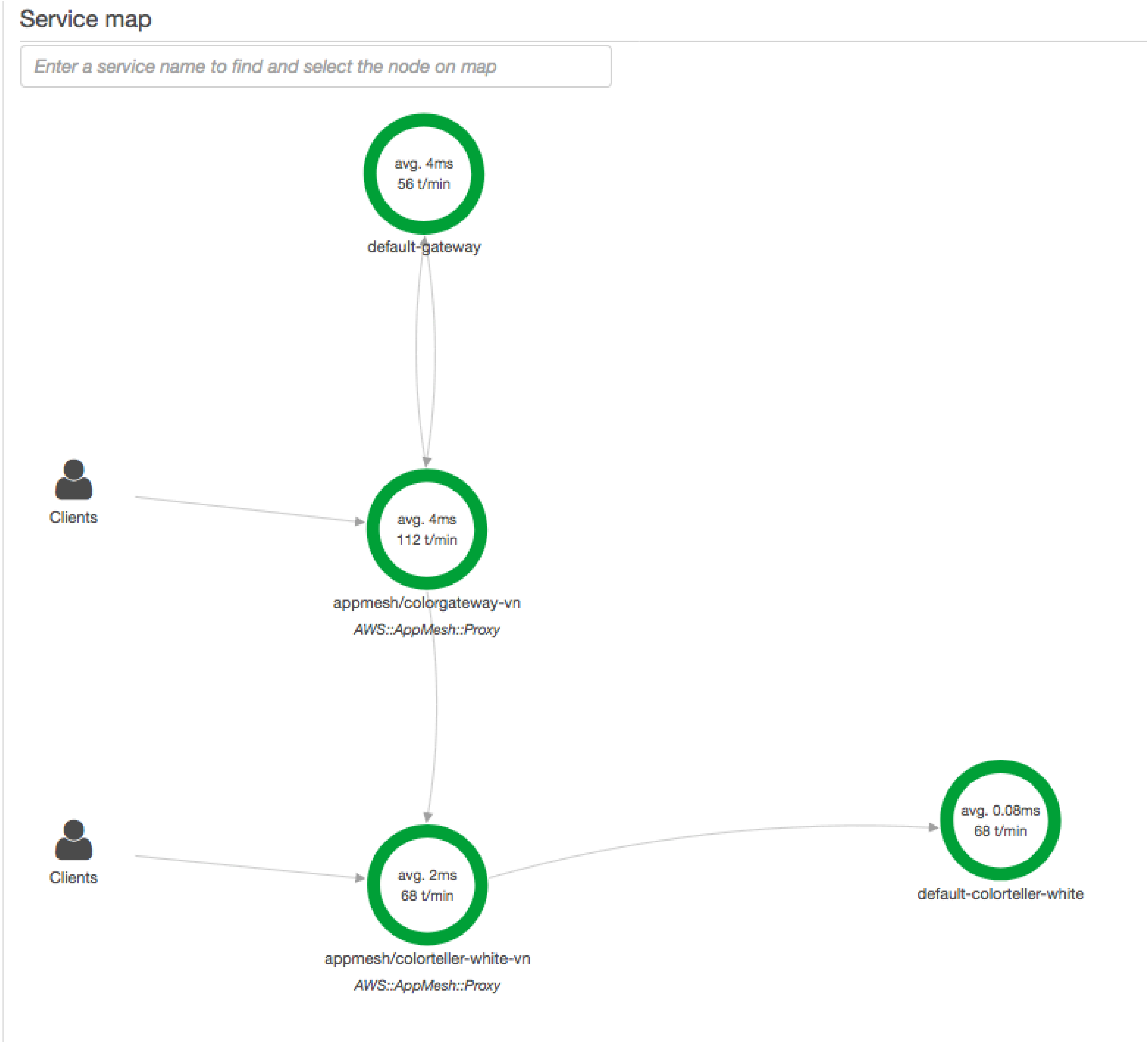
If you use API Gateway to generate a Java SDK for your API, you can instrument the SDK client by adding a request handler with the client builder, in the same way that you would manually instrument an AWS SDK client. See Tracing AWS SDK calls with the X-Ray SDK for Java for instructions.

Amazon EC2 and AWS App Mesh

AWS X-Ray integrates with AWS App Mesh to manage Envoy proxies for microservices. App Mesh provides a version of Envoy that you can configure to send trace data to the X-Ray daemon running in a container of the same task or pod. X-Ray supports tracing with the following App Mesh compatible services:

- Amazon Elastic Container Service (Amazon ECS)
- Amazon Elastic Kubernetes Service (Amazon EKS)
- Amazon Elastic Compute Cloud (Amazon EC2)

Use the following instructions to learn how to enable X-Ray tracing through App Mesh.



To configure the Envoy proxy to send data to X-Ray, set the `ENABLE_ENVOY_XRAY_TRACING` environment variable in its container definition.

Note

The App Mesh version of Envoy does not currently send traces based on configured sampling rules. Instead, it uses a fixed sampling rate of 5% for Envoy version 1.16.3 or newer, or a 50% sampling rate for Envoy versions prior to 1.16.3.

Example Envoy container definition for Amazon ECS

```
{
      "name": "envoy",
      "image": "public.ecr.aws/appmesh/aws-appmesh-envoy:envoy-version",
      "essential": true,
      "environment": [
        {
          "name": "APPMESH_VIRTUAL_NODE_NAME",
          "value": "mesh/myMesh/virtualNode/myNode"
        },
        {
          "name": "ENABLE_ENVOY_XRAY_TRACING",
          "value": "1"
        }
      ],
      "healthCheck": {
        "command": [
          "CMD-SHELL",
          "curl -s http://localhost:9901/server_info | cut -d' ' -f3 | grep -q live"
        ],
        "startPeriod": 10,
        "interval": 5,
        "timeout": 2,
        "retries": 3
      }
```

Note

To learn more about available Envoy region addresses, see Envoy image in the AWS App Mesh User Guide.

For details on running the X-Ray daemon in a container, see Running the X-Ray daemon on Amazon ECS. For a sample application that includes a service mesh, microservice, Envoy proxy, and X-Ray daemon, deploy the `colorapp` sample in the App Mesh Examples GitHub repository.

Learn More

- Getting Started with AWS App Mesh
- Getting Started with AWS App Mesh and Amazon ECS

AWS App Runner and X-Ray

AWS App Runner is an AWS service that provides a fast, simple, and cost-effective way to deploy from source code or a container image directly to a scalable and secure web application in the AWS Cloud. You don't need to learn new technologies, decide which compute service to use, or know how to provision and configure AWS resources. See What is AWS App Runner for more information.

AWS App Runner sends traces to X-Ray by integrating with the AWS Distro for OpenTelemetry (ADOT). Use ADOT SDKs to collect trace data for your containerized applications, and use X-Ray to analyze and gain insights into your instrumented application. For more information, see Tracing for your App Runner application with X-Ray.

Logging X-Ray API calls with AWS CloudTrail

AWS X-Ray is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service. CloudTrail captures all API calls for X-Ray as events. The calls captured include calls from the X-Ray console and code calls to the X-Ray API operations. Using the information collected by CloudTrail, you can determine the request that was made to X-Ray, the IP address from which the request was made, when it was made, and additional details.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root user or user credentials.
- Whether the request was made on behalf of an IAM Identity Center user.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

CloudTrail is active in your AWS account when you create the account and you automatically have access to the CloudTrail **Event history**. The CloudTrail **Event history** provides a viewable, searchable, downloadable, and immutable record of the past 90 days of recorded management events in an AWS Region. For more information, see Working with CloudTrail Event history in the *AWS CloudTrail User Guide*. There are no CloudTrail charges for viewing the **Event history**.

For an ongoing record of events in your AWS account past 90 days, create a trail or a CloudTrail Lake event data store.

CloudTrail trails

A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. All trails created using the AWS Management Console are multi-Region. You can create a single-Region or a multi-Region trail by using the AWS CLI. Creating a multi-Region trail is recommended because you capture activity in all AWS Regions in your account. If you create a single-Region trail, you can view only the events logged in the trail's AWS Region. For more information about trails, see Creating a trail for your AWS account and Creating a trail for an organization in the *AWS CloudTrail User Guide*.

You can deliver one copy of your ongoing management events to your Amazon S3 bucket at no charge from CloudTrail by creating a trail, however, there are Amazon S3 storage charges. For more information about CloudTrail pricing, see AWS CloudTrail Pricing. For information about Amazon S3 pricing, see Amazon S3 Pricing.

CloudTrail Lake event data stores

CloudTrail Lake lets you run SQL-based queries on your events. CloudTrail Lake converts existing events in row-based JSON format to Apache ORC format. ORC is a columnar storage format that is optimized for fast retrieval of data. Events are aggregated into *event data stores*, which are immutable collections of events based on criteria that you select by applying advanced event selectors. The selectors that you apply to an event data store control which events persist and are available for you to query. For more information about CloudTrail Lake, see Working with AWS CloudTrail Lake in the *AWS CloudTrail User Guide*.

CloudTrail Lake event data stores and queries incur costs. When you create an event data store, you choose the pricing option you want to use for the event data store. The pricing option determines the cost for ingesting and storing events, and the default and maximum retention period for the event data store. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

Topics

- X-Ray management events in CloudTrail
- X-Ray data events in CloudTrail
- X-Ray event examples

X-Ray management events in CloudTrail

AWS X-Ray integrates with AWS CloudTrail to record API actions made by a user, a role, or an AWS service in X-Ray. You can use CloudTrail to monitor X-Ray API requests in real time and store logs in Amazon S3, Amazon CloudWatch Logs, and Amazon CloudWatch Events. X-Ray supports logging the following actions as events in CloudTrail log files:

Supported API Actions

- PutEncryptionConfig
- GetEncryptionConfig
- CreateGroup
- UpdateGroup
- DeleteGroup
- GetGroup
- GetGroups
- GetInsight
- GetInsightEvents
- GetInsightImpactGraph
- GetInsightSummaries
- GetSamplingStatisticSummaries

X-Ray data events in CloudTrail

Data events provide information about the resource operations performed on or in a resource (for example, PutTraceSegments, which uploads segment documents to X-Ray).

These are also known as data plane operations. Data events are often high-volume activities. By default, CloudTrail doesn’t log data events. The CloudTrail **Event history** doesn't record data events.

Additional charges apply for data events. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

You can log data events for the X-Ray resource types by using the CloudTrail console, AWS CLI, or CloudTrail API operations. For more information about how to log data events, see Logging data events with the AWS Management Console and Logging data events with the AWS Command Line Interface in the *AWS CloudTrail User Guide*.

The following table lists the X-Ray resource types for which you can log data events. The **Data event type (console)** column shows the value to choose from the **Data event type** list on the CloudTrail console. The **resources.type value** column shows the `resources.type` value, which you would specify when configuring advanced event selectors using the AWS CLI or CloudTrail APIs. The **Data APIs logged to CloudTrail** column shows the API calls logged to CloudTrail for the resource type.

Data event type (console)	resources.type value	Data APIs logged to CloudTrail
X-Ray trace	`AWS::XRay::Trace`	• PutTraceSegments • GetTraceSummaries • GetTraceGraph • GetServiceGraph • BatchGetTraces • GetTimeSeriesServiceStatistics • PutTelemetryRecords • GetSamplingTargets

You can configure advanced event selectors to filter on the `eventName` and `readOnly` fields to log only those events that are important to you. However, you cannot select events by adding the `resources.ARN` field selector, because X-Ray traces do not have ARNs. For more information about these fields, see AdvancedFieldSelector in the *AWS CloudTrail API Reference*. The following is an example of how to run the `put-event-selectors` AWS CLI command to log data events on a CloudTrail trail. You must run the command in or specify the Region in which the trail was created; otherwise, the operation returns an `InvalidHomeRegionException` exception.

```
aws cloudtrail put-event-selectors --trail-name myTrail --advanced-event-selectors \
'{
   "AdvancedEventSelectors": [
      {
         "FieldSelectors": [
            { "Field": "eventCategory", "Equals": ["Data"] },
            { "Field": "resources.type", "Equals": ["AWS::XRay::Trace"] },
            { "Field": "eventName", "Equals":
 ["PutTraceSegments","GetSamplingTargets"] }
         ],
         "Name": "Log X-Ray PutTraceSegments and GetSamplingTargets data events"
      }
   ]
}'
```

X-Ray event examples

Management event example, `GetEncryptionConfig`

The following is an example of the X-Ray GetEncryptionConfig log entry in CloudTrail.

Example

```
{
    "eventVersion"=>"1.05",
    "userIdentity"=>{
        "type"=>"AssumedRole",
        "principalId"=>"AROAJVHBZWD3DN6CI2MHM:MyName",
        "arn"=>"arn:aws:sts::123456789012:assumed-role/MyRole/MyName",
        "accountId"=>"123456789012",
        "accessKeyId"=>"AKIAIOSFODNN7EXAMPLE",
        "sessionContext"=>{
            "attributes"=>{
                "mfaAuthenticated"=>"false",
                "creationDate"=>"2023-7-01T00:24:36Z"
            },
            "sessionIssuer"=>{
                "type"=>"Role",
                "principalId"=>"AROAJVHBZWD3DN6CI2MHM",
                "arn"=>"arn:aws:iam::123456789012:role/MyRole",
                "accountId"=>"123456789012",
                "userName"=>"MyRole"
            }
        }
    },
    "eventTime"=>"2023-7-01T00:24:36Z",
    "eventSource"=>"xray.amazonaws.com",
    "eventName"=>"GetEncryptionConfig",
    "awsRegion"=>"us-east-2",
    "sourceIPAddress"=>"33.255.33.255",
    "userAgent"=>"aws-sdk-ruby2/2.11.19 ruby/2.3.1 x86_64-linux",
    "requestParameters"=>nil,
    "responseElements"=>nil,
    "requestID"=>"3fda699a-32e7-4c20-37af-edc2be5acbdb",
    "eventID"=>"039c3d45-6baa-11e3-2f3e-e5a036343c9f",
    "eventType"=>"AwsApiCall",
    "recipientAccountId"=>"123456789012"
}
```

Data event example, `PutTraceSegments`

The following is an example of the X-Ray PutTraceSegments data event log entry in CloudTrail.

Example

```
{
  "eventVersion": "1.09",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AROAWYXPW54Y4NEXAMPLE:i-0dzz2ac111c83zz0z",
    "arn": "arn:aws:sts::012345678910:assumed-role/my-service-role/
i-0dzz2ac111c83zz0z",
    "accountId": "012345678910",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AROAWYXPW54Y4NEXAMPLE",
        "arn": "arn:aws:iam::012345678910:role/service-role/my-service-role",
        "accountId": "012345678910",
        "userName": "my-service-role"
      },
      "attributes": {
        "creationDate": "2024-01-22T17:34:11Z",
        "mfaAuthenticated": "false"
      },
      "ec2RoleDelivery": "2.0"
    }
  },
  "eventTime": "2024-01-22T18:22:05Z",
  "eventSource": "xray.amazonaws.com",
  "eventName": "PutTraceSegments",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "198.51.100.0",
  "userAgent": "aws-sdk-ruby3/3.190.0 md/internal ua/2.0 api/xray#1.0.0 os/linux md/
x86_64 lang/ruby#2.7.8 md/2.7.8 cfg/retry-mode#legacy",
  "requestParameters": {
    "traceSegmentDocuments": [
      "trace_id:1-00zzz24z-EXAMPLE4f4e41754c77d0000",
      "trace_id:1-00zzz24z-EXAMPLE4f4e41754c77d0000",
      "trace_id:1-00zzz24z-EXAMPLE4f4e41754c77d0001",
      "trace_id:1-00zzz24z-EXAMPLE4f4e41754c77d0002"
    ]
  },
  "responseElements": {
    "unprocessedTraceSegments": []
  },
  "requestID": "5zzzzz64-acbd-46ff-z544-451a3ebcb2f8",
  "eventID": "4zz51z7z-77f9-44zz-9bd7-6c8327740f2e",
  "readOnly": false,
  "resources": [
    {
      "type": "AWS::XRay::Trace"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": false,
  "recipientAccountId": "012345678910",
  "eventCategory": "Data",
  "tlsDetails": {
    "tlsVersion": "TLSv1.2",
    "cipherSuite": "ZZZZZ-RSA-AAA128-GCM-SHA256",
    "clientProvidedHostHeader": "example.us-west-2.xray.cloudwatch.aws.dev"
  }
}
```

CloudWatch integration with X-Ray

AWS X-Ray integrates with CloudWatch Application Signals, CloudWatch RUM, and CloudWatch Synthetics to make it easier to monitor the health of your applications. Enable your application for Application Signals to monitor and troubleshoot the operational health of your services, client pages, Synthetics canaries, and service dependencies.

By correlating CloudWatch metrics, logs, and X-Ray traces, the X-Ray trace map provides an end-to-end view of your services to help you quickly pinpoint performance bottlenecks and identify impacted users.

With CloudWatch RUM, you can perform real user monitoring to collect and view client-side data about your web application performance from actual user sessions in near-real time. With AWS X-Ray and CloudWatch RUM, you can analyze and debug the request path starting from end users of your application through downstream AWS managed services. This helps you identify latency trends and errors that impact your end users.

Topics

- CloudWatch RUM and AWS X-Ray
- Debugging CloudWatch synthetics canaries using X-Ray

CloudWatch RUM and AWS X-Ray

With Amazon CloudWatch RUM, you can perform real user monitoring to collect and view client-side data about your web application performance from actual user sessions in near-real time. With AWS X-Ray and CloudWatch RUM, you can analyze and debug the request path starting from end users of your application through downstream AWS managed services. This helps you identify latency trends and errors that impact your end users.

After you turn on X-Ray tracing of user sessions, CloudWatch RUM adds an X-Ray trace header to allowed HTTP requests, and records an X-Ray segment for allowed HTTP requests. You can then see traces and segments from these user sessions in the X-Ray and CloudWatch consoles, including the X-Ray trace map.

> **Note**
>
> CloudWatch RUM doesn't integrate with X-Ray sampling rules. Instead, choose a sampling percentage when you set up your application to use CloudWatch RUM. Traces sent from CloudWatch RUM might incur additional costs. For more information, see AWS X-Ray pricing.

By default, client-side traces sent from CloudWatch RUM aren't connected to server-side traces. To connect client-side traces with server-side traces, configure the CloudWatch RUM web client to add an X-Ray trace header to these HTTP requests.

⚠ **Warning**

Configuring the CloudWatch RUM web client to add an X-Ray trace header to HTTP requests can cause cross-origin resource sharing (CORS) to fail. To avoid this, add the `X-Amzn-Trace-Id` HTTP header to the list of allowed headers on your downstream service's CORS configuration. If you are using API Gateway as your downstream, see Enabling CORS for a REST API resource. We strongly recommend that you test your application before adding a client-side X-Ray trace header in a production environment. For more information, see the CloudWatch RUM web client documentation.

For more information about real user monitoring in CloudWatch, see Use CloudWatch RUM. To set up your application to use CloudWatch RUM, including tracing user sessions with X-Ray, see Set up an application to use CloudWatch RUM.

Debugging CloudWatch synthetics canaries using X-Ray

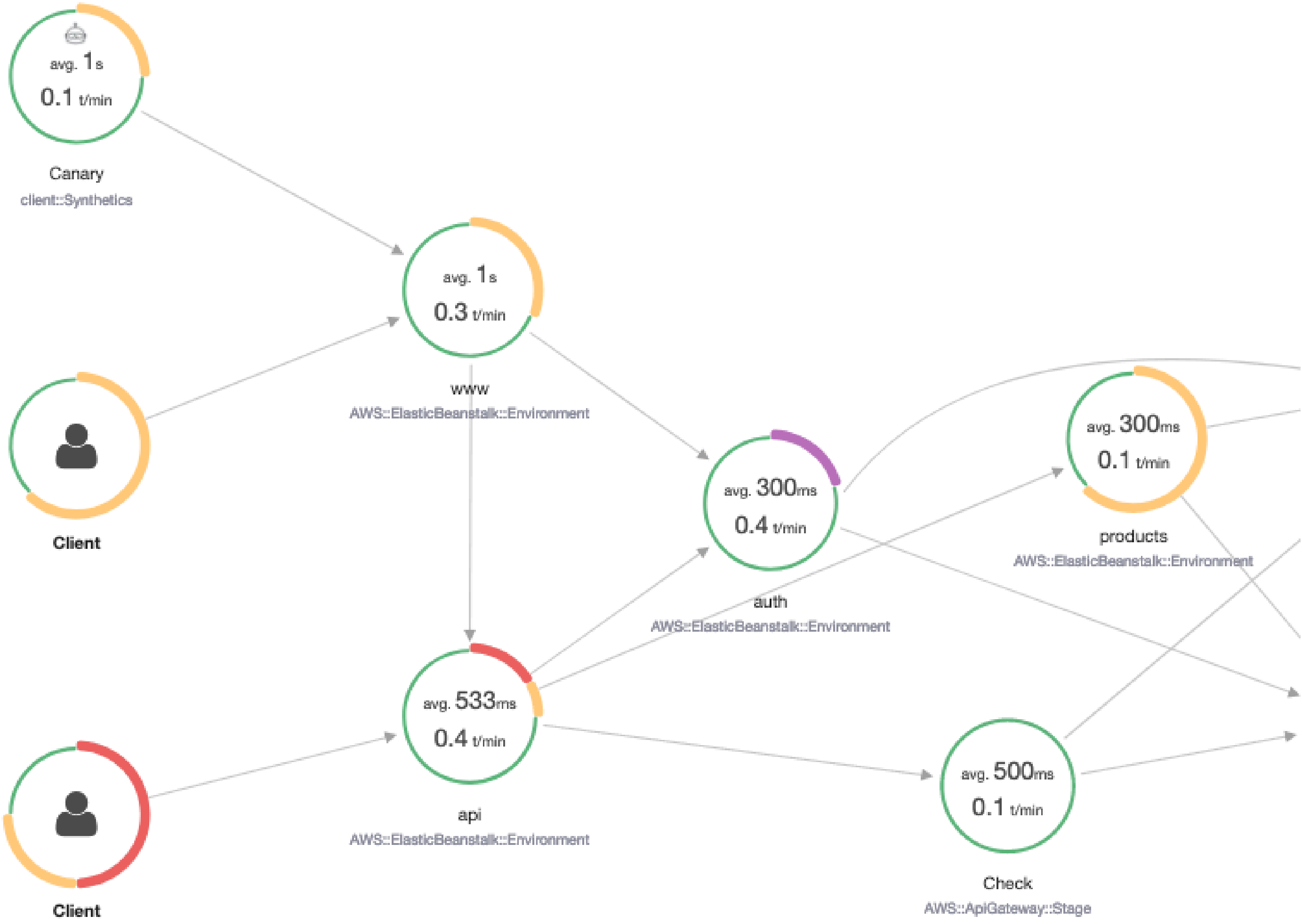
CloudWatch Synthetics is a fully managed service that enables you to monitor your endpoints and APIs using scripted canaries that run 24 hours per day, once per minute.

You can customize canary scripts to check for changes in:

- Availability
- Latency
- Transactions
- Broken or dead links
- Step-by-step task completions
- Page load errors
- Load Latencies for UI assets
- Complex wizard flows
- Checkout flows in your application

Canaries follow the same routes and perform the same actions and behaviors as your customers, and continually verify the customer experience.

To learn more about setting up Synthetics tests, see Using Synthetics to Create and Manage Canaries.



The following examples show common use cases for debugging issues that your Synthetics canaries raise. Each example demonstrates a key strategy for debugging using either the trace map or the X-Ray Analytics console.

For more information about how to read and interact with the trace map, see Viewing the Service Map.

For more information about how to read and interact with the X-Ray Analytics console, see Interacting with the AWS X-Ray Analytics Console.

Topics

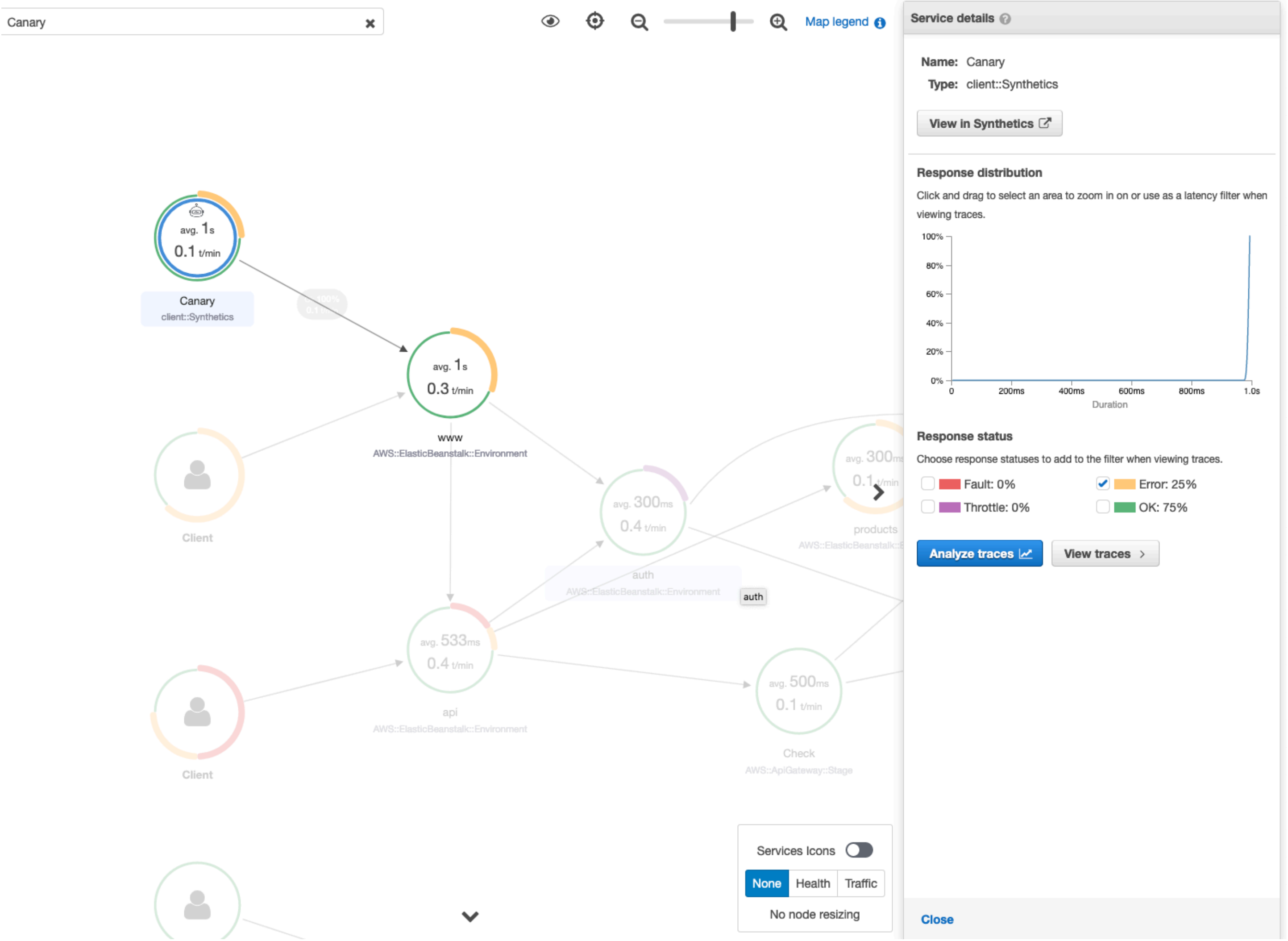
- View canaries with increased error reporting in the trace map
- Use trace details maps for individual traces to view each request in detail

- Determine the root cause of ongoing failures in upstream and downstream services
- Identify performance bottlenecks and trends
- Compare latency and error or fault rates before and after changes
- Determine the required canary coverage for all APIs and URLs
- Use groups to focus on synthetics tests

View canaries with increased error reporting in the trace map

To see which canaries have an increase in errors, faults, throttling rates, or slow response times within your X-Ray trace map, you can highlight Synthetics canary client nodes using the `Client::Synthetic` filter. Clicking a node displays the response time distribution of the entire request. Clicking an edge between two nodes shows details about the requests that traveled that connection. You can also view "remote" inferred nodes for related downstream services in your trace map.

When you click the Synthetics node, there is a **View in Synthetics** button on side panel which redirects you to the Synthetics console where you can check the canary details.



Service details

Name: Canary

Type: client::Synthetics

Response distribution

Click and drag to select an area to zoom in on or use as a latency filter when viewing traces.

Response status

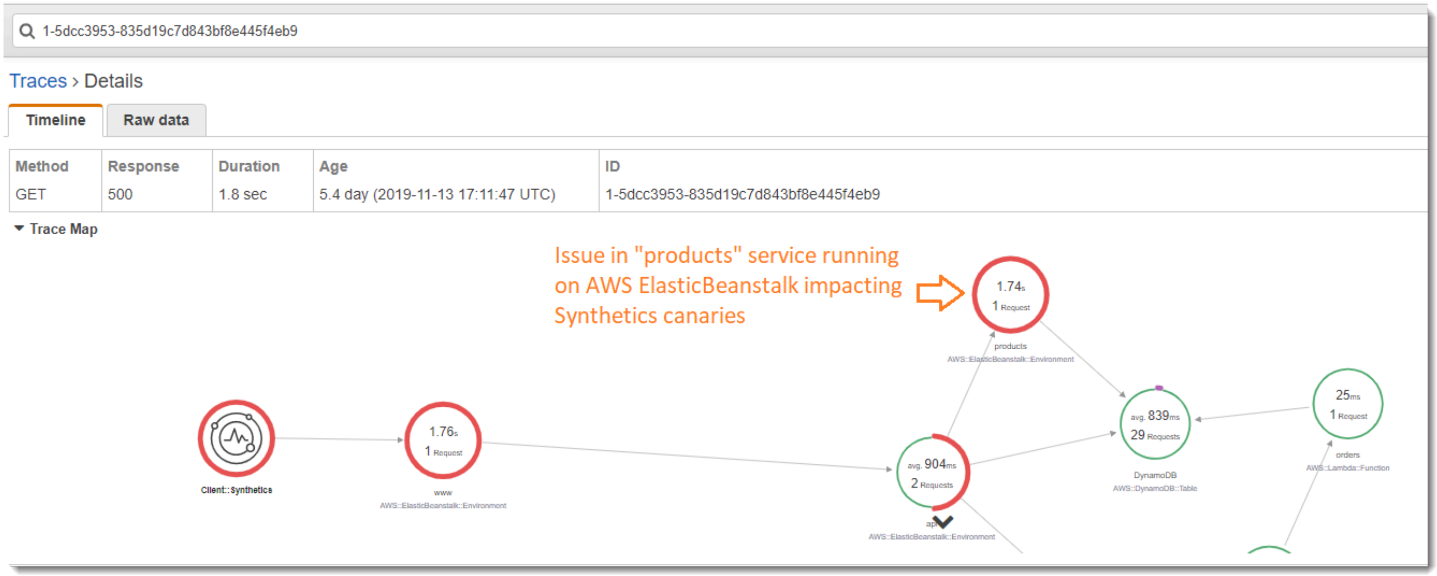
Choose response statuses to add to the filter when viewing traces.

□ 🟥 Fault: 0% ☑ 🟧 Error: 25%

□ 🟪 Throttle: 0% □ 🟩 OK: 75%

Use trace details maps for individual traces to view each request in detail

To determine which service results in the most latency or is causing an error, invoke the trace details map by selecting the trace in the trace map. Individual trace details maps display the end-to-end path of a single request. Use this to understand the services invoked, and visualize the upstream and downstream services.



1-5dcc3953-835d19c7d843bf8e445f4eb9

Traces › Details

Timeline | Raw data

Method	Response	Duration	Age	ID
GET	500	1.8 sec	5.4 day (2019-11-13 17:11:47 UTC)	1-5dcc3953-835d19c7d843bf8e445f4eb9

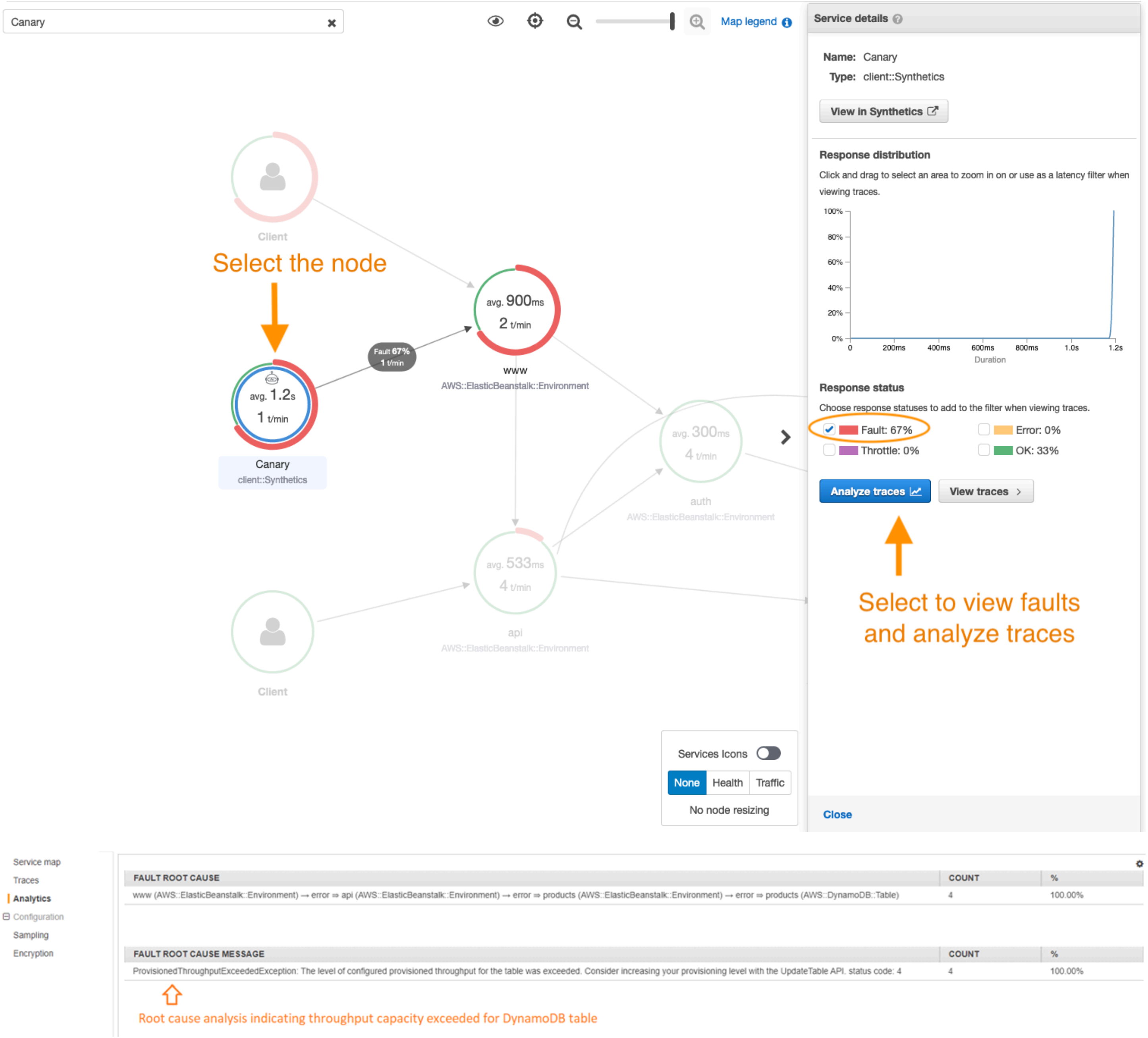
▾ Trace Map

Issue in "products" service running on AWS ElasticBeanstalk impacting Synthetics canaries

Determine the root cause of ongoing failures in upstream and downstream services

Once you receive a CloudWatch alarm for failures in a Synthetics canary, use the statistical modeling on trace data in X-Ray to determine the probable root cause of the issue within the X-Ray Analytics console. In the Analytics console, the **Response Time Root Cause** table shows recorded entity paths. X-Ray determines which path in your trace is the most likely cause for the response time. The format indicates a hierarchy of entities that are encountered, ending in a response time root cause.

The following example shows that the Synthetics test for API “XXX” running on API Gateway is failing due to a throughput capacity exception from the Amazon DynamoDB table.



Select the node

Name: Canary
Type: client::Synthetics

Response distribution

Click and drag to select an area to zoom in on or use as a latency filter when viewing traces.

Response status

Choose response statuses to add to the filter when viewing traces.

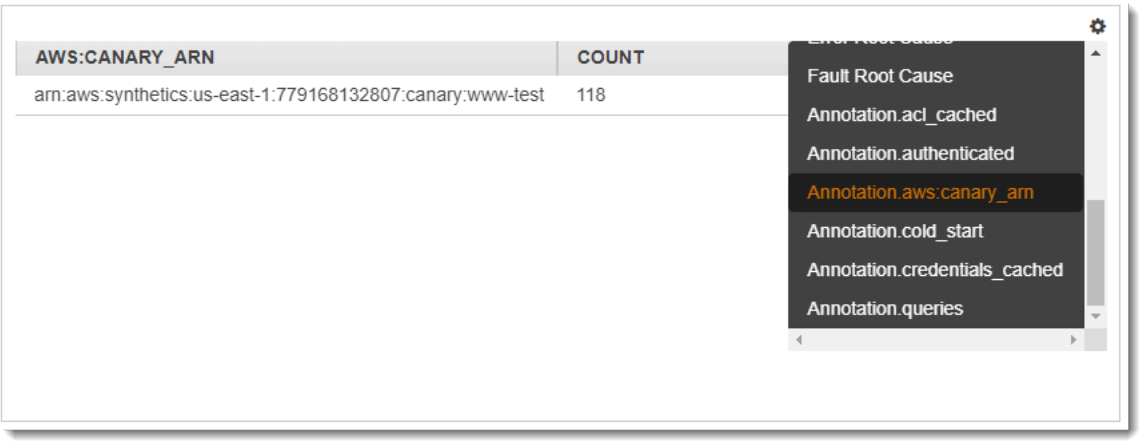
☑ 🟥 Fault: 67%   □ 🟧 Error: 0%
□ 🟪 Throttle: 0%   □ 🟩 OK: 33%

Select to view faults and analyze traces

FAULT ROOT CAUSE	COUNT	%
www (AWS::ElasticBeanstalk::Environment) → error ⇒ api (AWS::ElasticBeanstalk::Environment) → error ⇒ products (AWS::ElasticBeanstalk::Environment) → error ⇒ products (AWS::DynamoDB::Table)	4	100.00%

FAULT ROOT CAUSE MESSAGE	COUNT	%
ProvisionedThroughputExceededException: The level of configured provisioned throughput for the table was exceeded. Consider increasing your provisioning level with the UpdateTable API. status code: 4	4	100.00%

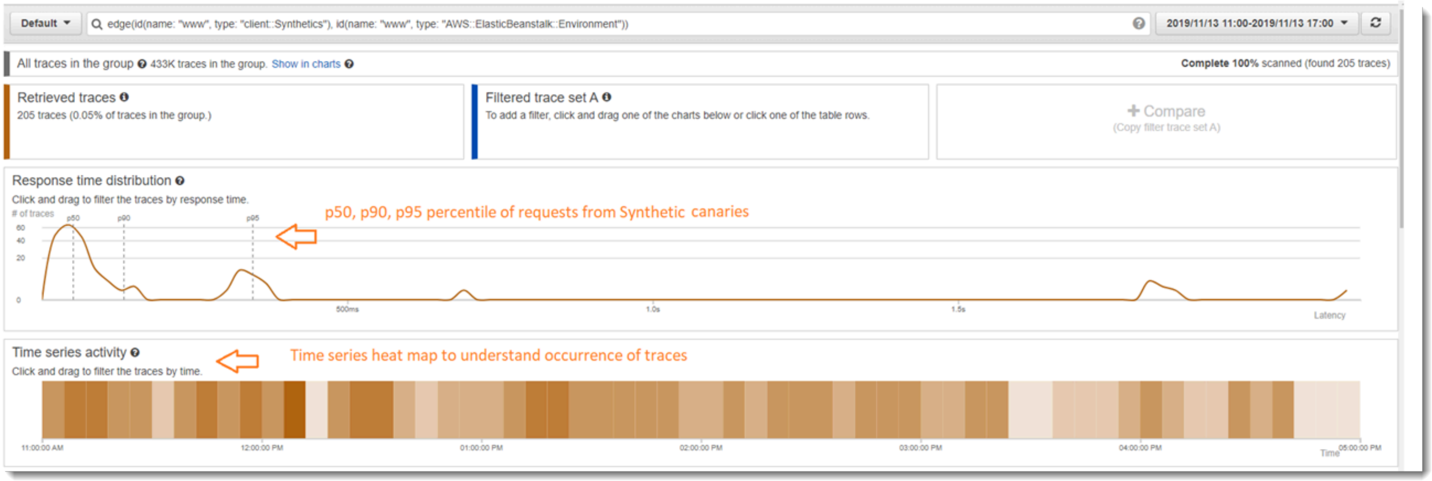
Root cause analysis indicating throughput capacity exceeded for DynamoDB table



AWS:CANARY_ARN	COUNT
arn:aws:synthetics:us-east-1:779168132807:canary:www-test	118

Identify performance bottlenecks and trends

You can view trends in the performance of your endpoint over time using continuous traffic from your Synthetics canaries to populate a trace details map over a period of time.



Default

edge(id(name: "www", type: "client::Synthetics"), id(name: "www", type: "AWS::ElasticBeanstalk::Environment"))

2019/11/13 11:00-2019/11/13 17:00

All traces in the group 433K traces in the group. Show in charts

Complete 100% scanned (found 205 traces)

Retrieved traces

205 traces (0.05% of traces in the group.)

Filtered trace set A

To add a filter, click and drag one of the charts below or click one of the table rows.

+ Compare

(Copy filter trace set A)

Response time distribution

Click and drag to filter the traces by response time.

p50, p90, p95 percentile of requests from Synthetic canaries

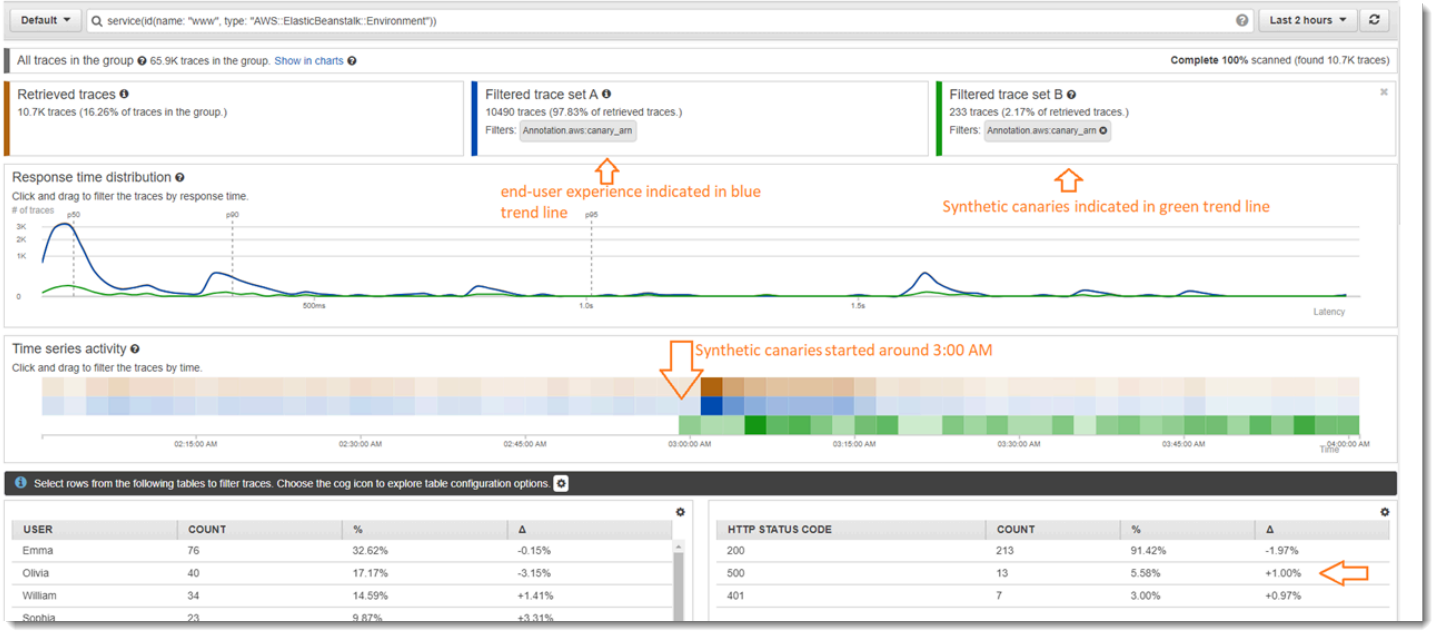
Time series activity

Click and drag to filter the traces by time.

Time series heat map to understand occurrence of traces

Compare latency and error or fault rates before and after changes

Pinpoint the time a change occurred to correlate that change to an increase in issues caught by your canaries. Use the X-Ray Analytics console to define the before and after time ranges as different trace sets, creating a visual differentiation in the response time distribution.



service(id(name: "www", type: "AWS::ElasticBeanstalk::Environment"))

All traces in the group 65.9K traces in the group. Show in charts

Complete 100% scanned (found 10.7K traces)

Retrieved traces
10.7K traces (16.26% of traces in the group.)

Filtered trace set A
10490 traces (97.83% of retrieved traces.)
Filters: Annotation.aws:canary_arn

Filtered trace set B
233 traces (2.17% of retrieved traces.)
Filters: Annotation.aws:canary_arn

Response time distribution
Click and drag to filter the traces by response time.

end-user experience indicated in blue trend line

Synthetic canaries indicated in green trend line

Time series activity
Click and drag to filter the traces by time.

Synthetic canaries started around 3:00 AM

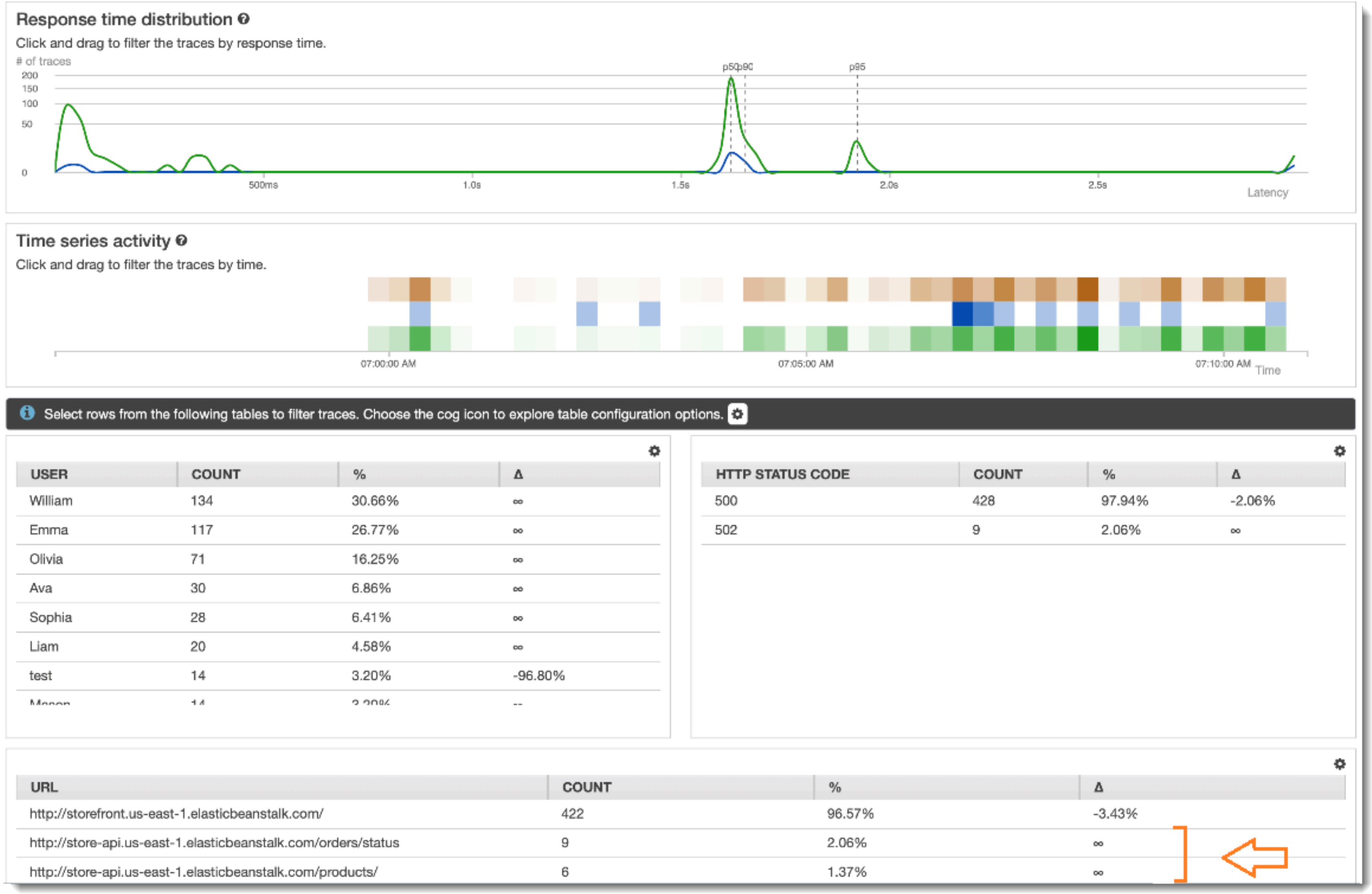
Select rows from the following tables to filter traces. Choose the cog icon to explore table configuration options.

USER	COUNT	%	Δ
Emma	76	32.62%	-0.15%
Olivia	40	17.17%	-3.15%
William	34	14.59%	+1.41%
Sophia	23	9.87%	+3.31%

HTTP STATUS CODE	COUNT	%	Δ
200	213	91.42%	-1.97%
500	13	5.58%	+1.00%
401	7	3.00%	+0.97%

Determine the required canary coverage for all APIs and URLs

Use X-Ray Analytics to compare the experience of canaries with the users. The UI below shows a blue trend line for canaries and a green line for the users. You can also identify that two out of the three URLs don't have canary tests.



Response time distribution

Click and drag to filter the traces by response time.

Time series activity

Click and drag to filter the traces by time.

Select rows from the following tables to filter traces. Choose the cog icon to explore table configuration options.

USER	COUNT	%	Δ
William	134	30.66%	∞
Emma	117	26.77%	∞
Olivia	71	16.25%	∞
Ava	30	6.86%	∞
Sophia	28	6.41%	∞
Liam	20	4.58%	∞
test	14	3.20%	-96.80%
Mason	14	3.20%	--

HTTP STATUS CODE	COUNT	%	Δ
500	428	97.94%	-2.06%
502	9	2.06%	∞

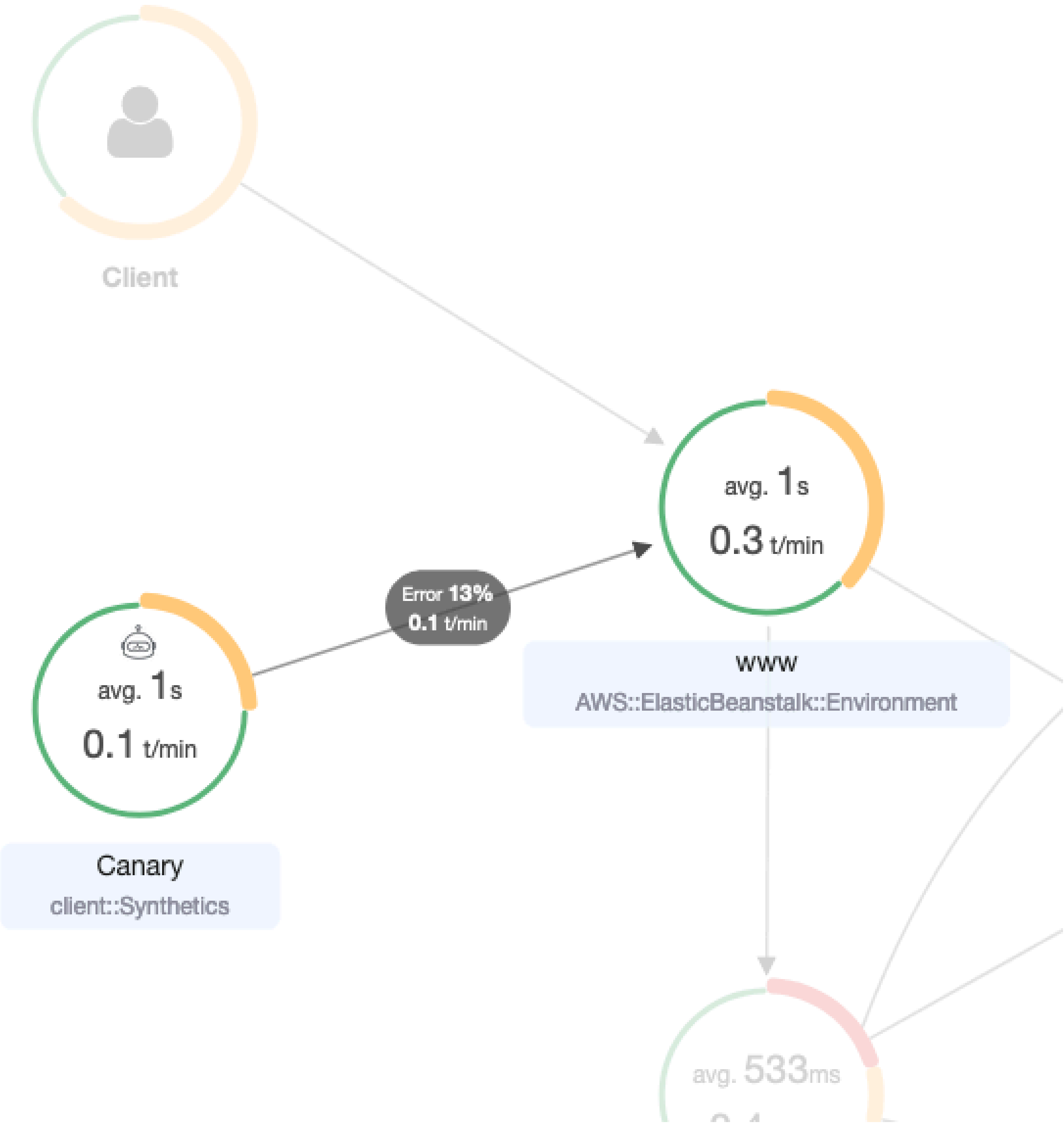
URL	COUNT	%	Δ
http://storefront.us-east-1.elasticbeanstalk.com/	422	96.57%	-3.43%
http://store-api.us-east-1.elasticbeanstalk.com/orders/status	9	2.06%	∞
http://store-api.us-east-1.elasticbeanstalk.com/products/	6	1.37%	∞

Use groups to focus on synthetics tests

You can create an X-Ray group using a filter expression to focus on a certain set of workflows, such as a Synthetics tests for application “www” running on AWS Elastic Beanstalk. Use the complex keywords `service()` and `edge()` to filter through services and edges.

Example Group filter expression

```
"edge(id(name: "www", type: "client::Synthetics"), id(name: "www", type:
 "AWS::ElasticBeanstalk::Environment"))"
```



AWS Elastic Beanstalk and AWS X-Ray

Important

End of support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the timeline, see X-Ray SDK and daemon end of support timeline and for information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

AWS Elastic Beanstalk platforms include the X-Ray daemon. You can run the daemon by setting an option in the Elastic Beanstalk console or with a configuration file.

On the Java SE platform, you can use a Buildfile file to build your application with Maven or Gradle on-instance. The X-Ray SDK for Java and AWS SDK for Java are available from Maven, so you can deploy only your application code and build on-instance to avoid bundling and uploading all of your dependencies.

You can use Elastic Beanstalk environment properties to configure the X-Ray SDK. The method that Elastic Beanstalk uses to pass environment properties to your application varies by platform. Use the X-Ray SDK's environment variables or system properties depending on your platform.

- **Node.js platform** – Use environment variables
- **Java SE platform** – Use environment variables
- **Tomcat platform** – Use system properties

For more information, see Configuring AWS X-Ray Debugging in the AWS Elastic Beanstalk Developer Guide.

Elastic Load Balancing and AWS X-Ray

Elastic Load Balancing application load balancers add a trace ID to incoming HTTP requests in a header named `X-Amzn-Trace-Id`.

```
X-Amzn-Trace-Id: Root=1-5759e988-bd862e3fe1be46a994272793
```

X-Ray trace ID format

An X-Ray `trace_id` consists of three numbers separated by hyphens. For example, `1-58406520-a006649127e371903a2de979`. This includes:

- The version number, which is 1.
- The time of the original request in Unix epoch time using **8 hexadecimal digits**.

  For example, 10:00AM December 1st, 2016 PST in epoch time is `1480615200` seconds or `58406520` in hexadecimal digits.
- A globally unique 96-bit identifier for the trace in **24 hexadecimal digits**.

Load balancers do not send data to X-Ray, and do not appear as a node on your service map.

For more information, see Request Tracing for Your Application Load Balancer in the Elastic Load Balancing Developer Guide.

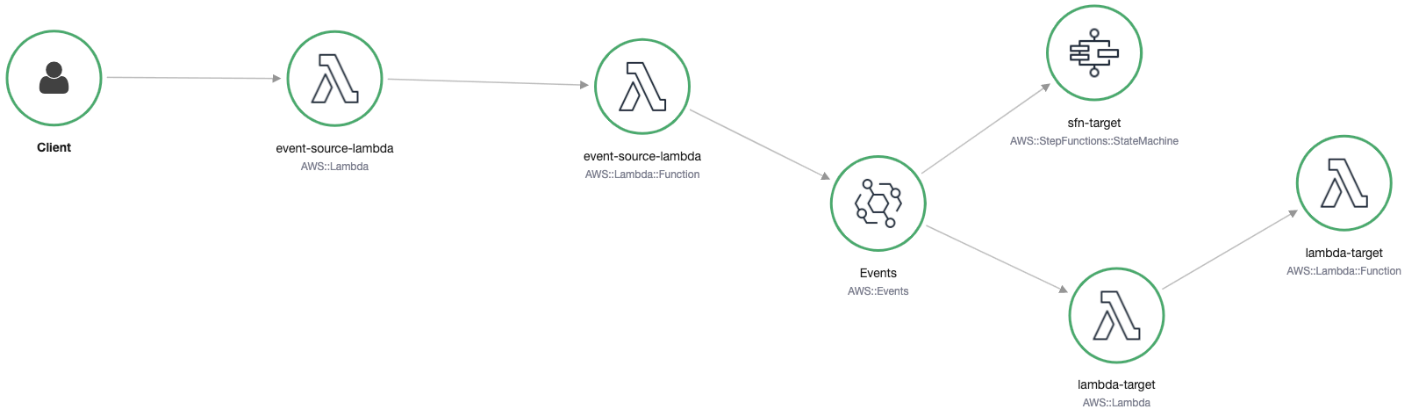
Amazon EventBridge and AWS X-Ray

AWS X-Ray integrates with Amazon EventBridge to trace events that are passed through EventBridge. If a service that is instrumented with the X-Ray SDK sends events to EventBridge, the trace context is propagated to downstream event targets within the tracing header. The X-Ray SDK automatically picks up the tracing header and applies it to any subsequent instrumentation. This continuity enables users to trace, analyze, and debug throughout downstream services and provides a more complete view of their system.

For more information, see EventBridge X-Ray Integration in the *EventBridge User Guide*.

Viewing source and targets on the X-Ray service map

The X-Ray trace map displays an EventBridge event node that connects source and target services, as in the following example:



Propagate the trace context to event targets

The X-Ray SDK enables the EventBridge event source to propagate trace context to downstream event targets. The following language-specific examples demonstrate calling EventBridge from a Lambda function on which active tracing is enabled:

Java

Add the necessary dependencies for X-Ray:

- AWS X-Ray SDK for Java
- AWS X-Ray Recorder SDK for 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.xray.AWSXRay;
import com.amazonaws.services.eventbridge.AmazonEventBridge;
import com.amazonaws.services.eventbridge.AmazonEventBridgeClientBuilder;
import com.amazonaws.services.eventbridge.model.PutEventsRequest;
import com.amazonaws.services.eventbridge.model.PutEventsRequestEntry;
import com.amazonaws.services.eventbridge.model.PutEventsResult;
import com.amazonaws.services.eventbridge.model.PutEventsResultEntry;
import com.amazonaws.xray.handlers.TracingHandler;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.lang.StringBuilder;
import java.util.Map;
import java.util.List;
import java.util.Date;
import java.util.Collections;

/*
   Add the necessary dependencies for XRay:
   https://mvnrepository.com/artifact/com.amazonaws/aws-java-sdk-xray
   https://mvnrepository.com/artifact/com.amazonaws/aws-xray-recorder-sdk-aws-sdk
*/
public class Handler implements RequestHandler<SQSEvent, String>{
  private static final Logger logger = LoggerFactory.getLogger(Handler.class);

  /*
    build EventBridge client
  */
  private static final AmazonEventBridge eventsClient =
 AmazonEventBridgeClientBuilder
          .standard()
          // instrument the EventBridge client with the XRay Tracing Handler.
          // the AWSXRay globalRecorder will retrieve the tracing-context
          // from the lambda function and inject it into the HTTP header.
          // be sure to enable 'active tracing' on the lambda function.
          .withRequestHandlers(new TracingHandler(AWSXRay.getGlobalRecorder()))
          .build();

  @Override
  public String handleRequest(SQSEvent event, Context context)
  {
    PutEventsRequestEntry putEventsRequestEntry0 = new PutEventsRequestEntry();
    putEventsRequestEntry0.setTime(new Date());
    putEventsRequestEntry0.setSource("my-lambda-function");
    putEventsRequestEntry0.setDetailType("my-lambda-event");
    putEventsRequestEntry0.setDetail("{\"lambda-source\":\"sqs\"}");
    PutEventsRequest putEventsRequest = new PutEventsRequest();
    putEventsRequest.setEntries(Collections.singletonList(putEventsRequestEntry0));
    // send the event(s) to EventBridge
    PutEventsResult putEventsResult = eventsClient.putEvents(putEventsRequest);
    try {
      logger.info("Put Events Result: {}", putEventsResult);
    } catch(Exception e) {
      e.getStackTrace();
    }
    return "success";
  }
}
```

Python

Add the following dependency to your requirements.txt file:

```
aws-xray-sdk==2.4.3
```

```
import boto3
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

# apply the XRay handler to all clients.
patch_all()
```

```
client = boto3.client('events')

def lambda_handler(event, context):
    response = client.put_events(
        Entries=[
            {
                'Source': 'foo',
                'DetailType': 'foo',
                'Detail': '{\"foo\": \"foo\"}'
            },
        ]
    )
    return response
```

Go

```
package main

import (
  "context"
  "github.com/aws/aws-lambda-go/lambda"
  "github.com/aws/aws-lambda-go/events"
  "github.com/aws/aws-sdk-go/aws/session"
  "github.com/aws/aws-xray-sdk-go/xray"
  "github.com/aws/aws-sdk-go/service/eventbridge"
  "fmt"
)

var client = eventbridge.New(session.New())


func main() {
 //Wrap the eventbridge client in the AWS XRay tracer
  xray.AWS(client.Client)
  lambda.Start(handleRequest)
}

func handleRequest(ctx context.Context, event events.SQSEvent) (string, error) {
  _, err := callEventBridge(ctx)
  if err != nil {
    return "ERROR", err
  }
  return "success", nil
}


func callEventBridge(ctx context.Context) (string, error) {
    entries := make([]*eventbridge.PutEventsRequestEntry, 1)
    detail := "{ \"foo\": \"foo\"}"
    detailType := "foo"
    source := "foo"
    entries[0] = &eventbridge.PutEventsRequestEntry{
        Detail: &detail,
        DetailType: &detailType,
        Source: &source,
    }

  input := &eventbridge.PutEventsInput{
     Entries: entries,
  }

  // Example sending a request using the PutEventsRequest method.
  resp, err := client.PutEventsWithContext(ctx, input)

  success := "yes"
  if err == nil { // resp is now filled
      success = "no"
      fmt.Println(resp)
  }
  return success, err
}
```

Node.js

```
const AWSXRay = require('aws-xray-sdk')
//Wrap the aws-sdk client in the AWS XRay tracer
const AWS = AWSXRay.captureAWS(require('aws-sdk'))
const eventBridge = new AWS.EventBridge()

exports.handler = async (event) => {

  let myDetail = { "name": "Alice" }

  const myEvent = {
    Entries: [{
      Detail: JSON.stringify({ myDetail }),
      DetailType: 'myDetailType',
      Source: 'myApplication',
      Time: new Date
    }]
  }

  // Send to EventBridge
  const result = await eventBridge.putEvents(myEvent).promise()

  // Log the result
  console.log('Result: ', JSON.stringify(result, null, 2))

}
```

C#

Add the following X-Ray packages to your C# dependencies:

```
<PackageReference Include="AWSXRayRecorder.Core" Version="2.6.2" />
<PackageReference Include="AWSXRayRecorder.Handlers.AwsSdk" Version="2.7.2" />
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Amazon;
using Amazon.Util;
using Amazon.Lambda;
using Amazon.Lambda.Model;
using Amazon.Lambda.Core;
using Amazon.EventBridge;
using Amazon.EventBridge.Model;
using Amazon.Lambda.SQSEvents;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.AwsSdk;
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]

namespace blankCsharp
```

```
{
  public class Function
  {
    private static AmazonEventBridgeClient eventClient;

    static Function() {
      initialize();
    }

    static async void initialize() {
      //Wrap the AWS SDK clients in the AWS XRay tracer
      AWSSDKHandler.RegisterXRayForAllServices();
      eventClient = new AmazonEventBridgeClient();
    }

    public async Task<PutEventsResponse> FunctionHandler(SQSEvent invocationEvent,
 ILambdaContext context)
    {
      PutEventsResponse response;
      try
      {
        response = await callEventBridge();
      }
      catch (AmazonLambdaException ex)
      {
        throw ex;
      }

      return response;
    }

    public static async Task<PutEventsResponse> callEventBridge()
    {
      var request = new PutEventsRequest();
      var entry = new PutEventsRequestEntry();
      entry.DetailType = "foo";
      entry.Source = "foo";
      entry.Detail = "{\"instance_id\":\"A\"}";
      List<PutEventsRequestEntry> entries = new List<PutEventsRequestEntry>();
      entries.Add(entry);
      request.Entries = entries;
      var response = await eventClient.PutEventsAsync(request);
      return response;
    }
    }
}
```

AWS Lambda and AWS X-Ray

Important

End of support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the timeline, see X-Ray SDK and daemon end of support timeline and for information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can use AWS X-Ray to trace your AWS Lambda functions. Lambda runs the X-Ray daemon and records a segment with details about invoking and running the function. For further instrumentation, you can bundle the X-Ray SDK with your function to record outgoing calls and add annotations and metadata.

If your Lambda function is called by another instrumented service, Lambda traces requests that have already been sampled without any additional configuration. The upstream service can be an instrumented web application or another Lambda function. Your service can invoke the function directly with an instrumented AWS SDK client, or by calling an API Gateway API with an instrumented HTTP client.

AWS X-Ray supports tracing event-driven applications using AWS Lambda and Amazon SQS. Use the CloudWatch console to see a connected view of each request as it's queued with Amazon SQS and processed by a downstream Lambda function. Traces from upstream message producers are automatically linked to traces from downstream Lambda consumer nodes, creating an end-to-end view of the application. For more information, see tracing event-driven applications.

Note

If you have traces enabled for a downstream Lambda function, you must also have traces enabled for the root Lambda function that calls the downstream function in order for the downstream function to generate traces.

If your Lambda function runs on a schedule, or is invoked by a service that is not instrumented, you can configure Lambda to sample and record invocations with active tracing.

To configure X-Ray integration on an AWS Lambda function

1. Open the AWS Lambda console.
2. Select **Functions** from the left navigation bar.
3. Choose your function.
4. On the **Configuration** tab, scroll down to the **Additional monitoring tools** card. You can also find this card by selecting **Monitoring and operations tools** on the left navigation pane.
5. Select **Edit**.
6. Under **AWS X-Ray**, enable **Active tracing**.

On runtimes with a corresponding X-Ray SDK, Lambda also runs the X-Ray daemon.

X-Ray SDKs on Lambda

- **X-Ray SDK for Go** – Go 1.7 and newer runtimes
- **X-Ray SDK for Java** – Java 8 runtime
- **X-Ray SDK for Node.js** – Node.js 4.3 and newer runtimes
- **X-Ray SDK for Python** – Python 2.7, Python 3.6, and newer runtimes
- **X-Ray SDK for .NET** – .NET Core 2.0 and newer runtimes

To use the X-Ray SDK on Lambda, bundle it with your function code each time you create a new version. You can instrument your Lambda functions with the same methods that you use to instrument applications running on other services. The primary difference is that you don't use the SDK to instrument incoming requests, make sampling decisions, and create segments.

The other difference between instrumenting Lambda functions and web applications is that the segment that Lambda creates and sends to X-Ray can't be modified by your function code. You can create subsegments and record annotations and metadata on them, but you can't add annotations and metadata to the parent segment.

For more information, see Using AWS X-Ray in the *AWS Lambda Developer Guide*.

AWS Step Functions and AWS X-Ray

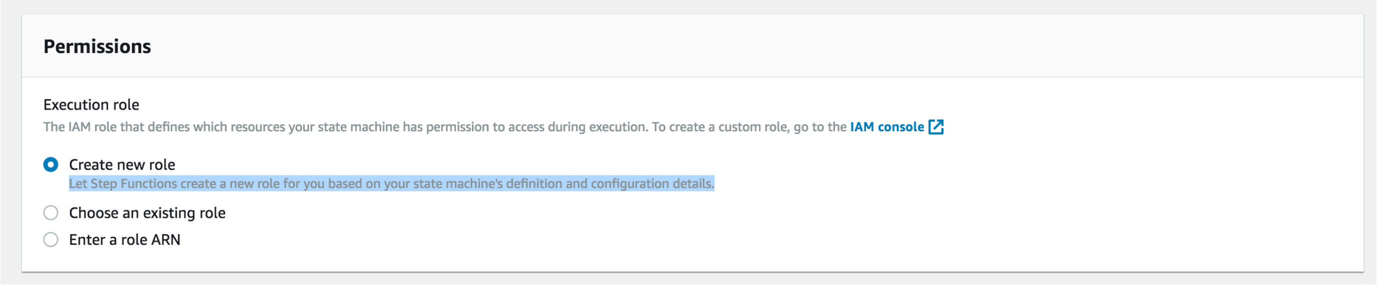
AWS X-Ray integrates with AWS Step Functions to trace and analyze requests for Step Functions. You can visualize the components of your state machine, identify performance bottlenecks, and troubleshoot requests that resulted in an error. For more information, see AWS X-Ray and Step Functions in the AWS Step Functions Developer Guide.

To enable X-Ray tracing when creating a new state machine

1. Open the Step Functions console at https://console.aws.amazon.com/states/.
2. Choose **Create a state machine**.
3. On the **Define state machine** page, choose either **Author with code snippets** or **Start with a template**. If you choose to run a sample project, you can't enable X-Ray tracing during creation. Instead, enable X-Ray tracing after you create your state machine.
4. Choose **Next**.
5. On the **Specify details** page, configure your state machine.
6. Choose **Enable X-Ray tracing**.

To enable X-Ray tracing in an existing state machine

1. In the Step Functions console, select the state machine for which you want to enable tracing.
2. Choose **Edit**.
3. Choose **Enable X-Ray tracing**.
4. (Optional) Auto-generate a new role for your state machine to include X-Ray permissions by choosing **Create new role** from the Permissions window.



Permissions

Execution role

The IAM role that defines which resources your state machine has permission to access during execution. To create a custom role, go to the **IAM console**

● Create new role
Let Step Functions create a new role for you based on your state machine's definition and configuration details.

○ Choose an existing role

○ Enter a role ARN

5. Choose **Save**.

Note

When you create a new state machine, it's automatically traced if the request is sampled and tracing is enabled in an upstream service such as Amazon API Gateway or AWS Lambda. For any existing state machine not configured through the console, for example through an AWS CloudFormation template, check that you have an IAM policy that grants sufficient permissions to enable X-Ray traces.

Instrumenting your application for AWS X-Ray

Important

End of support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the timeline, see X-Ray SDK and daemon end of support timeline and for information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

Instrumenting your application involves sending trace data for incoming and outbound requests and other events within your application, along with metadata about each request. There are several different instrumentation options you can choose from or combine, based on your particular requirements:

- *Auto instrumentation* – instrument your application with zero code changes, typically via configuration changes, adding an auto-instrumentation agent, or other mechanisms.
- *Library instrumentation* – make minimal application code changes to add pre-built instrumentation targeting specific libraries or frameworks, such as the AWS SDK, Apache HTTP clients, or SQL clients.
- *Manual instrumentation* – add instrumentation code to your application at each location where you want to send trace information.

There are several SDKs, agents, and tools that can be used to instrument your application for X-Ray tracing.

Topics

- Instrumenting your application with the AWS Distro for OpenTelemetry
- Instrumenting your application with AWS X-Ray SDKs
- Choosing between the AWS Distro for OpenTelemetry and X-Ray SDKs

Instrumenting your application with the AWS Distro for OpenTelemetry

The AWS Distro for OpenTelemetry (ADOT) is an AWS distribution based on the Cloud Native Computing Foundation (CNCF) OpenTelemetry project. OpenTelemetry provides a single set of open source APIs, libraries, and agents to collect distributed traces and metrics. This toolkit is a distribution of upstream OpenTelemetry components including SDKs, auto-instrumentation agents, and collectors that are tested, optimized, secured, and supported by AWS.

With ADOT, engineers can instrument their applications once and send correlated metrics and traces to multiple AWS monitoring solutions including Amazon CloudWatch, AWS X-Ray, and Amazon OpenSearch Service.

Using X-Ray with ADOT requires two components: an *OpenTelemetry SDK* enabled for use with X-Ray, and the *AWS Distro for OpenTelemetry Collector* enabled for use with X-Ray. For more information about using the AWS Distro for OpenTelemetry with AWS X-Ray and other AWS services, see the AWS Distro for OpenTelemetry Documentation.

For more information about language support and usage, see AWS Observability on GitHub.

> **Note**
>
> You can now use the CloudWatch agent to collect metrics, logs and traces from Amazon EC2 instances and on-premise servers. CloudWatch agent version 1.300025.0 and later can collect traces from OpenTelemetry or X-Ray client SDKs, and send them to X-Ray. Using the CloudWatch agent instead of the AWS Distro for OpenTelemetry (ADOT) Collector or X-Ray daemon to collect traces can help you reduce the number of agents that you manage. See the CloudWatch agent topic in the CloudWatch User Guide for more information.

ADOT includes the following:

- AWS Distro for OpenTelemetry Go
- AWS Distro for OpenTelemetry Java
- AWS Distro for OpenTelemetry JavaScript
- AWS Distro for OpenTelemetry Python
- AWS Distro for OpenTelemetry .NET

ADOT currently includes auto-instrumentation support for Java and Python. In addition, ADOT enables auto-instrumentation of AWS Lambda functions and their downstream requests using Java, Node.js, and Python runtimes, via ADOT Managed Lambda Layers.

ADOT SDKs for Java and Go support X-Ray centralized sampling rules. If you need support for X-Ray sampling rules in other languages, consider using an AWS X-Ray SDK.

> **Note**
>
> You can send now send W3C trace IDs to X-Ray. By default, traces that are created with OpenTelemetry have a trace ID format that's based on the W3C Trace Context specification. This is different from the format for trace IDs that are created using an X-Ray SDK or by AWS services that are integrated with X-Ray. To ensure that trace IDs in W3C format are accepted by X-Ray, you must use AWS X-Ray Exporter version 0.86.0 or later, which is included with ADOT Collector version 0.34.0 and later. Previous versions of the exporter validate trace ID timestamps, which might cause W3C trace IDs to be rejected.

Instrumenting your application with AWS X-Ray SDKs

AWS X-Ray includes a set of language-specific SDKs for instrumenting your application to send traces to X-Ray. Each X-Ray SDK provides the following:

- *Interceptors* to add to your code to trace incoming HTTP requests
- *Client handlers* to instrument AWS SDK clients that your application uses to call other AWS services
- An *HTTP client* to instrument calls to other internal and external HTTP web services

X-Ray SDKs also support instrumenting calls to SQL databases, automatic AWS SDK client instrumentation, and other features. Instead of sending trace data directly to X-Ray, the SDK sends JSON segment documents to a daemon process listening for UDP traffic. The X-Ray daemon buffers segments in a queue and uploads them to X-Ray in batches.

The following language-specific SDKs are provided:

- AWS X-Ray SDK for Go
- AWS X-Ray SDK for Java

- AWS X-Ray SDK for Node.js
- AWS X-Ray SDK for Python
- AWS X-Ray SDK for .NET
- AWS X-Ray SDK for Ruby

X-Ray currently includes auto-instrumentation support for Java.

Choosing between the AWS Distro for OpenTelemetry and X-Ray SDKs

The SDKs included with X-Ray are part of a tightly integrated instrumentation solution offered by AWS. The AWS Distro for OpenTelemetry is part of a broader industry solution in which X-Ray is only one of many tracing solutions. You can implement end-to-end tracing in X-Ray using either approach, but it's important to understand the differences in order to determine the most useful approach for you.

We recommend instrumenting your application with the AWS Distro for OpenTelemetry if you need the following:

- The ability to send traces to multiple different tracing back ends without having to re-instrument your code
- Support for a large number of library instrumentations for each language, maintained by the OpenTelemetry community
- Fully managed Lambda layers that package everything you need to collect telemetry data, without requiring code changes when using Java, Python, or Node.js

> **Note**
>
> AWS Distro for OpenTelemetry offers a simpler getting started experience for instrumenting your Lambda functions. However, due to the flexibility OpenTelemetry offers, your Lambda function will require additional memory and invocations may experience cold start latency increases, which can lead to additional charges. If you're optimizing for low-latency and do not require OpenTelemetry's advanced capabilities such as dynamically configurable back end destinations, you may want to use the AWS X-Ray SDK to instrument your application.

We recommend choosing an X-Ray SDK for instrumenting your application if you need the following:

- A tightly integrated single-vendor solution
- Integration with X-Ray centralized sampling rules, including the ability to configure sampling rules from the X-Ray console and automatically use them across multiple hosts, when using Node.js, Python, Ruby, or .NET

Transaction Search

Transaction Search is an interactive analytics experience you can use to get complete visibility of your application transaction spans. Spans are the fundamental units of operation in a distributed trace and represent specific actions or tasks in an application or system. Every span records details about a particular segment of the transaction. These details include start and end times, duration, and associated metadata, which can include business attributes like customer IDs and order IDs. Spans are arranged in a parent-child hierarchy. This heirarchy forms a complete trace mapping the flow of a transaction across different components or services.

For more information, see Transaction Search.

OpenTelemetry Protocol (OTLP) Endpoint

OpenTelemetry is an open-source observability framework that provides IT teams with standardized protocols and tools for collecting and routing telemetry data. It delivers a unified format for instrumenting, generating, gathering, and exporting application telemetry data, such as metrics, logs, and traces to monitoring platforms for analysis and insights. By using OpenTelemetry, teams can avoid vendor lock-in, ensuring flexibility in their observability solutions.

You can use OpenTelemetry to directly send traces to an OpenTelemetry Protocol (OTLP) endpoint, and get out-of-the box application performance monitoring experiences in CloudWatch Application Signals.

For more information, see OpenTelemetry.

Working with Go

There are two ways to instrument your Go application to send traces to X-Ray:

- AWS Distro for OpenTelemetry Go – An AWS distribution that provides a set of open source libraries for sending correlated metrics and traces to multiple AWS monitoring solutions including Amazon CloudWatch, AWS X-Ray, and Amazon OpenSearch Service, via the AWS Distro for OpenTelemetry Collector.
- AWS X-Ray SDK for Go – A set of libraries for generating and sending traces to X-Ray via the X-Ray daemon.

For more information, see Choosing between the AWS Distro for OpenTelemetry and X-Ray SDKs.

AWS Distro for OpenTelemetry Go

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

With the AWS Distro for OpenTelemetry Go, you can instrument your applications once and send correlated metrics and traces to multiple AWS monitoring solutions including Amazon CloudWatch, AWS X-Ray, and Amazon OpenSearch Service. Using X-Ray with AWS Distro for OpenTelemetry requires two components: an *OpenTelemetry SDK* enabled for use with X-Ray, and the *AWS Distro for OpenTelemetry Collector* enabled for use with X-Ray.

To get started, see the AWS Distro for OpenTelemetry Go documentation.

For more information about using the AWS Distro for OpenTelemetry with AWS X-Ray and other AWS services, see AWS Distro for OpenTelemetry or the AWS Distro for OpenTelemetry Documentation.

For more information about language support and usage, see AWS Observability on GitHub.

AWS X-Ray SDK for Go

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The X-Ray SDK for Go is a set of libraries for Go applications that provide classes and methods for generating and sending trace data to the X-Ray daemon. Trace data includes information about incoming HTTP requests served by the application, and calls that the application makes to downstream services using the AWS SDK, HTTP clients, or an SQL database connector. You can also create segments manually and add debug information in annotations and metadata.

Download the SDK from its GitHub repository with `go get`:

```
$ go get -u github.com/aws/aws-xray-sdk-go/...
```

For web applications, start by using the `xray.Handler` function to trace incoming requests. The message handler creates a segment for each traced request, and completes the segment when the response is sent. While the segment is open you can use the SDK client's methods to add information to the segment and create subsegments to trace downstream calls. The SDK also automatically records exceptions that your application throws while the segment is open.

For Lambda functions called by an instrumented application or service, Lambda reads the tracing header and traces sampled requests automatically. For other functions, you can configure Lambda to sample and trace incoming requests. In either case, Lambda creates the segment and provides it to the X-Ray SDK.

Note

On Lambda, the X-Ray SDK is optional. If you don't use it in your function, your service map will still include a node for the Lambda service, and one for each Lambda function. By adding the SDK, you can instrument your function code to add subsegments to the function segment recorded by Lambda. See AWS Lambda and AWS X-Ray for more information.

Next, wrap your client with a call to the AWS function. This step ensures that X-Ray instruments calls to any client methods. You can also instrument calls to SQL databases.

After you start using the SDK, customize its behavior by configuring the recorder and middleware. You can add plugins to record data about the compute resources running your application, customize sampling behavior by defining sampling rules, and set the log level to see more or less information from the SDK in your application logs.

Record additional information about requests and the work that your application does in annotations and metadata. Annotations are simple key-value pairs that are indexed for use with filter expressions, so that you can search for traces that contain specific data. Metadata entries are less restrictive and can record entire objects and arrays — anything that can be serialized into JSON.

Annotations and Metadata

Annotations and metadata are arbitrary text that you add to segments with the X-Ray SDK. Annotations are indexed for use with filter expressions. Metadata are not indexed, but can be viewed in the raw segment with the X-Ray console or API. Anyone that you grant read access to X-Ray can view this data.

When you have a lot of instrumented clients in your code, a single request segment can contain a large number of subsegments, one for each call made with an instrumented client. You can organize and group subsegments by wrapping client calls in custom subsegments. You can create a custom subsegment for an entire function or any section of code, and record metadata and annotations on the subsegment instead of writing everything on the parent segment.

Requirements

The X-Ray SDK for Go requires Go 1.9 or later.

The SDK depends on the following libraries at compile and runtime:

- AWS SDK for Go version 1.10.0 or newer

These dependencies are declared in the SDK's `README.md` file.

Reference documentation

Once you have downloaded the SDK, build and host the documentation locally to view it in a web browser.

To view the reference documentation

1. Navigating to the `$GOPATH/src/github.com/aws/aws-xray-sdk-go` (Linux or Mac) directory or the `%GOPATH%\src\github.com\aws\aws-xray-sdk-go` (Windows) folder
2. Run the `godoc` command.

```
$ godoc -http=:6060
```

3. Opening a browser at `http://localhost:6060/pkg/github.com/aws/aws-xray-sdk-go/`.

Configuring the X-Ray SDK for Go

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can specify the configuration for X-Ray SDK for Go through environment variables, by calling `Configure` with a `Config` object, or by assuming default values. Environment variables take precedence over `Config` values, which take precedence over any default value.

Sections

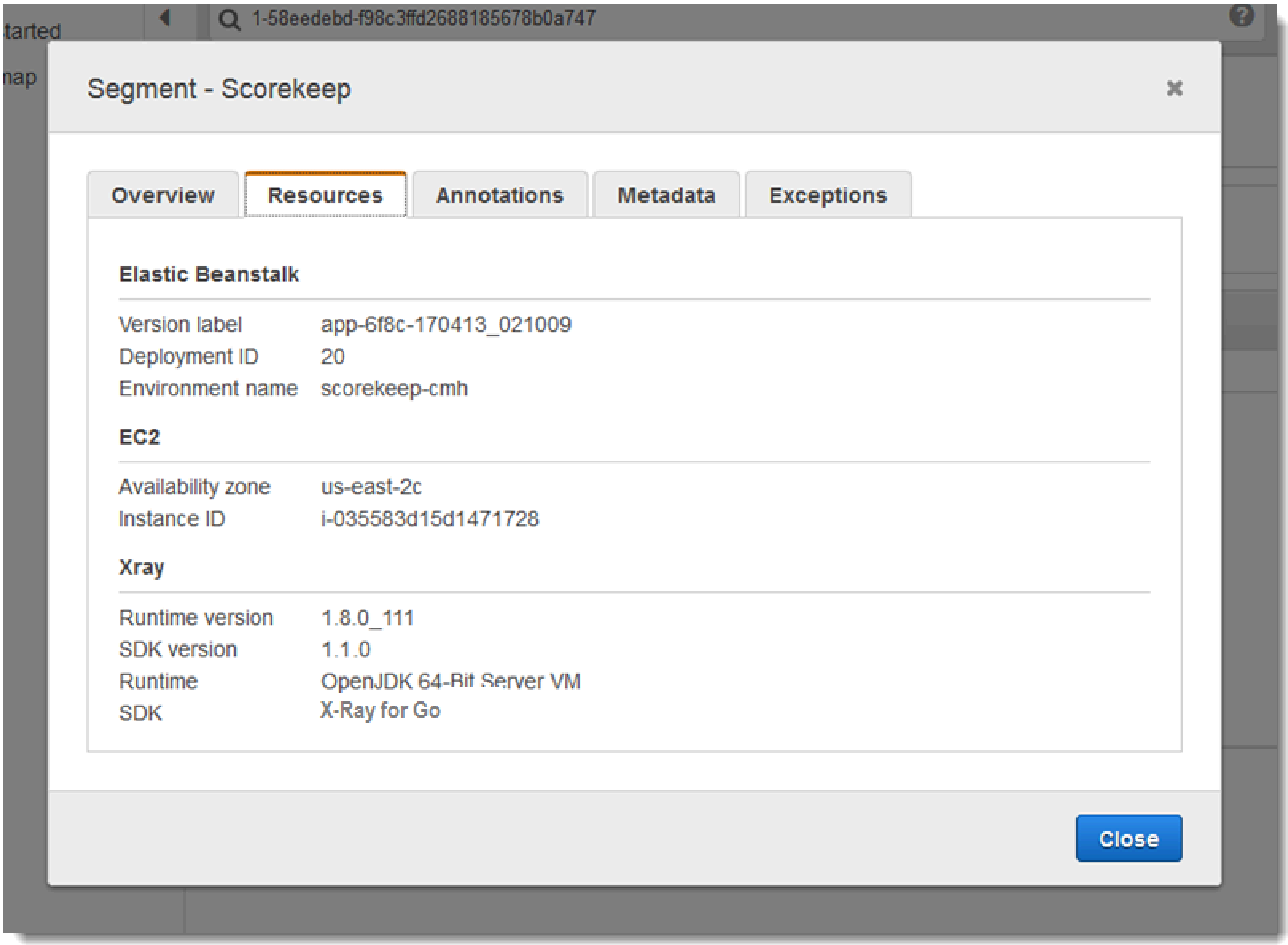
- Service plugins
- Sampling rules
- Logging
- Environment variables
- Using configure

Service plugins

Use `plugins` to record information about the service hosting your application.

Plugins

- Amazon EC2 – `EC2Plugin` adds the instance ID, Availability Zone, and the CloudWatch Logs Group.
- Elastic Beanstalk – `ElasticBeanstalkPlugin` adds the environment name, version label, and deployment ID.
- Amazon ECS – `ECSPlugin` adds the container ID.



Segment - Scorekeep

Elastic Beanstalk

Version label	app-6f8c-170413_021009
Deployment ID	20
Environment name	scorekeep-cmh

EC2

Availability zone	us-east-2c
Instance ID	i-035583d15d1471728

Xray

Runtime version	1.8.0_111
SDK version	1.1.0
Runtime	OpenJDK 64-Bit Server VM
SDK	X-Ray for Go

To use a plugin, import one of the following packages.

```
"github.com/aws/aws-xray-sdk-go/awsplugins/ec2"
"github.com/aws/aws-xray-sdk-go/awsplugins/ecs"
"github.com/aws/aws-xray-sdk-go/awsplugins/beanstalk"
```

Each plugin has an explicit `Init()` function call that loads the plugin.

Example ec2.Init()

```
import (
 "os"

 "github.com/aws/aws-xray-sdk-go/awsplugins/ec2"
 "github.com/aws/aws-xray-sdk-go/xray"
)

func init() {
  // conditionally load plugin
  if os.Getenv("ENVIRONMENT") == "production" {
    ec2.Init()
  }

  xray.Configure(xray.Config{
    ServiceVersion: "1.2.3",
  })
}
```

The SDK also uses plugin settings to set the `origin` field on the segment. This indicates the type of AWS resource that runs your application. When you use multiple plugins, the SDK uses the following resolution order to determine the origin: ElasticBeanstalk > EKS > ECS > EC2.

Sampling rules

The SDK uses the sampling rules you define in the X-Ray console to determine which requests to record. The default rule traces the first request each second, and five percent of any additional requests across all services sending traces to X-Ray. Create additional rules in the X-Ray console to customize the amount of data recorded for each of your applications.

The SDK applies custom rules in the order in which they are defined. If a request matches multiple custom rules, the SDK applies only the first rule.

> **Note**
>
> If the SDK can't reach X-Ray to get sampling rules, it reverts to a default local rule of the first request each second, and five percent of any additional requests per host. This can occur if the host doesn't have permission to call sampling APIs, or can't connect to the X-Ray daemon, which acts as a TCP proxy for API calls made by the SDK.

You can also configure the SDK to load sampling rules from a JSON document. The SDK can use local rules as a backup for cases where X-Ray sampling is unavailable, or use local rules exclusively.

Example sampling-rules.json

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

This example defines one custom rule and a default rule. The custom rule applies a five-percent sampling rate with no minimum number of requests to trace for paths under `/api/move/`. The default rule traces the first request each second and 10 percent of additional requests.

The disadvantage of defining rules locally is that the fixed target is applied by each instance of the recorder independently, instead of being managed by the X-Ray service. As you deploy more hosts, the fixed rate is multiplied, making it harder to control the amount of data recorded.

On AWS Lambda, you cannot modify the sampling rate. If your function is called by an instrumented service, calls that generated requests that were sampled by that service will be recorded by Lambda. If active tracing is enabled and no tracing header is present, Lambda makes the sampling decision.

To provide backup rules, point to the local sampling JSON file by using `NewCentralizedStrategyWithFilePath`.

Example main.go – Local sampling rule

```
s, _ := sampling.NewCentralizedStrategyWithFilePath("sampling.json") // path to local
 sampling json
xray.Configure(xray.Config{SamplingStrategy: s})
```

To use only local rules, point to the local sampling JSON file by using `NewLocalizedStrategyFromFilePath`.

Example main.go – Disable sampling

```
s, _ := sampling.NewLocalizedStrategyFromFilePath("sampling.json") // path to local
 sampling json
xray.Configure(xray.Config{SamplingStrategy: s})
```

Logging

Note

The `xray.Config{}` fields `LogLevel` and `LogFormat` are deprecated starting with version 1.0.0-rc.10.

X-Ray uses the following interface for logging. The default logger writes to `stdout` at `LogLevelInfo` and above.

```
type Logger interface {
 Log(level LogLevel, msg fmt.Stringer)
}

const (
 LogLevelDebug LogLevel = iota + 1
 LogLevelInfo
 LogLevelWarn
 LogLevelError
)
```

Example write to `io.Writer`

```
xray.SetLogger(xraylog.NewDefaultLogger(os.Stderr, xraylog.LogLevelError))
```

Environment variables

You can use environment variables to configure the X-Ray SDK for Go. The SDK supports the following variables.

- `AWS_XRAY_CONTEXT_MISSING` – Set to `RUNTIME_ERROR` to throw exceptions when your instrumented code attempts to record data when no segment is open.

  Valid Values

  - `RUNTIME_ERROR` – Throw a runtime exception.
  - `LOG_ERROR` – Log an error and continue (default).
  - `IGNORE_ERROR` – Ignore error and continue.

  Errors related to missing segments or subsegments can occur when you attempt to use an instrumented client in startup code that runs when no request is open, or in code that spawns a new thread.

- `AWS_XRAY_TRACING_NAME` – Set the service name that the SDK uses for segments.
- `AWS_XRAY_DAEMON_ADDRESS` – Set the host and port of the X-Ray daemon listener. By default, the SDK sends trace data to `127.0.0.1:2000`. Use this variable if you have configured the daemon to listen on a different port or if it is running on a different host.
- `AWS_XRAY_CONTEXT_MISSING` – Set the value to determine how the SDK handles missing context errors. Errors related to missing segments or subsegments can occur when you attempt to use an instrumented client in the startup code when no request is open, or in code that spawns a new thread.
  - `RUNTIME_ERROR` – By default, the SDK is set to throw a runtime exception.
  - `LOG_ERROR` – Set to log an error and continue.

Environment variables override equivalent values set in code.

Using configure

You can also configure the X-Ray SDK for Go using the `Configure` method. `Configure` takes one argument, a `Config` object, with the following, optional fields.

DaemonAddr

  This string specifies the host and port of the X-Ray daemon listener. If not specified, X-Ray uses the value of the `AWS_XRAY_DAEMON_ADDRESS` environment variable. If that value is not set, it uses "127.0.0.1:2000".

ServiceVersion

  This string specifies the version of the service. If not specified, X-Ray uses the empty string ("").

SamplingStrategy

This `SamplingStrategy` object specifies which of your application calls are traced. If not specified, X-Ray uses a `LocalizedSamplingStrategy`, which takes the strategy as defined in `xray/resources/DefaultSamplingRules.json`.

StreamingStrategy

This `StreamingStrategy` object specifies whether to stream a segment when **RequiresStreaming** returns **true**. If not specified, X-Ray uses a `DefaultStreamingStrategy` that streams a sampled segment if the number of subsegments is greater than 20.

ExceptionFormattingStrategy

This `ExceptionFormattingStrategy` object specifies how you want to handle various exceptions. If not specified, X-Ray uses a `DefaultExceptionFormattingStrategy` with an `XrayError` of type `error`, the error message, and stack trace.

Instrumenting incoming HTTP requests with the X-Ray SDK for Go

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can use the X-Ray SDK to trace incoming HTTP requests that your application serves on an EC2 instance in Amazon EC2, AWS Elastic Beanstalk, or Amazon ECS.

Use `xray.Handler` to instrument incoming HTTP requests. The X-Ray SDK for Go implements the standard Go library `http.Handler` interface in the `xray.Handler` class to intercept web requests. The `xray.Handler` class wraps the provided `http.Handler` with `xray.Capture` using the request's context, parsing the incoming headers, adding response headers if needed, and sets HTTP-specific trace fields.

When you use this class to handle HTTP requests and responses, the X-Ray SDK for Go creates a segment for each sampled request. This segment includes timing, method, and disposition of the HTTP request. Additional instrumentation creates subsegments on this segment.

> **Note**
>
> For AWS Lambda functions, Lambda creates a segment for each sampled request. See AWS Lambda and AWS X-Ray for more information.

The following example intercepts requests on port 8000 and returns "Hello!" as a response. It creates the segment `myApp` and instruments calls through any application.

Example main.go

```
func main() {
  http.Handle("/", xray.Handler(xray.NewFixedSegmentNamer("MyApp"),
 http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello!"))
  })))

  http.ListenAndServe(":8000", nil)
}
```

Each segment has a name that identifies your application in the service map. The segment can be named statically, or you can configure the SDK to name it dynamically based on the host header in the incoming request. Dynamic naming lets you group traces based on the domain name in the request, and apply a default name if the name doesn't match an expected pattern (for example, if the host header is forged).

> **Forwarded Requests**
>
> If a load balancer or other intermediary forwards a request to your application, X-Ray takes the client IP from the `X-Forwarded-For` header in the request instead of from the source IP in the IP packet. The client IP that is recorded for a forwarded request can be forged, so it should not be trusted.

When a request is forwarded, the SDK sets an additional field in the segment to indicate this. If the segment contains the field `x_forwarded_for` set to `true`, the client IP was taken from the `X-Forwarded-For` header in the HTTP request.

The handler creates a segment for each incoming request with an `http` block that contains the following information:

- **HTTP method** – GET, POST, PUT, DELETE, etc.
- **Client address** – The IP address of the client that sent the request.
- **Response code** – The HTTP response code for the completed request.
- **Timing** – The start time (when the request was received) and end time (when the response was sent).
- **User agent** — The `user-agent` from the request.
- **Content length** — The `content-length` from the response.

Configuring a segment naming strategy

AWS X-Ray uses a *service name* to identify your application and distinguish it from the other applications, databases, external APIs, and AWS resources that your application uses. When the X-Ray SDK generates segments for incoming requests, it records your application's service name in the segment's name field.

The X-Ray SDK can name segments after the hostname in the HTTP request header. However, this header can be forged, which could result in unexpected nodes in your service map. To prevent the SDK from naming segments incorrectly due to requests with forged host headers, you must specify a default name for incoming requests.

If your application serves requests for multiple domains, you can configure the SDK to use a dynamic naming strategy to reflect this in segment names. A dynamic naming strategy allows the SDK to use the hostname for requests that match an expected pattern, and apply the default name to requests that don't.

For example, you might have a single application serving requests to three subdomains– `www.example.com`, `api.example.com`, and `static.example.com`. You can use a dynamic naming strategy with the pattern `*.example.com` to identify segments for each subdomain with a different name, resulting in three service nodes on the service map. If your application receives requests with a hostname that doesn't match the pattern, you will see a fourth node on the service map with a fallback name that you specify.

To use the same name for all request segments, specify the name of your application when you create the handler, as shown in the previous section.

Note

You can override the default service name that you define in code with the AWS_XRAY_TRACING_NAME environment variable.

A dynamic naming strategy defines a pattern that hostnames should match, and a default name to use if the hostname in the HTTP request doesn't match the pattern. To name segments dynamically, use NewDynamicSegmentNamer to configure the default name and pattern to match.

Example main.go

If the hostname in the request matches the pattern *.example.com, use the hostname. Otherwise, use MyApp.

```
func main() {
  http.Handle("/", xray.Handler(xray.NewDynamicSegmentNamer("MyApp", "*.example.com"),
 http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello!"))
  })))

  http.ListenAndServe(":8000", nil)
}
```

Tracing AWS SDK calls with the X-Ray SDK for Go

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

When your application makes calls to AWS services to store data, write to a queue, or send notifications, the X-Ray SDK for Go tracks the calls downstream in subsegments. Traced AWS services and resources that you access within those services (for example, an Amazon S3 bucket or Amazon SQS queue), appear as downstream nodes on the trace map in the X-Ray console.

To trace AWS SDK clients, wrap the client object with the `xray.AWS()` call as shown in the following example.

Example main.go

```
var dynamo *dynamodb.DynamoDB
func main() {
  dynamo = dynamodb.New(session.Must(session.NewSession()))
  xray.AWS(dynamo.Client)
}
```

Then, when you use the AWS SDK client, use the `withContext` version of the call method, and pass it the `context` from the `http.Request` object passed to the handler.

Example main.go – AWS SDK call

```
func listTablesWithContext(ctx context.Context) {
  output := dynamo.ListTablesWithContext(ctx, &dynamodb.ListTablesInput{})
  doSomething(output)
}
```

For all services, you can see the name of the API called in the X-Ray console. For a subset of services, the X-Ray SDK adds information to the segment to provide more granularity in the service map.

For example, when you make a call with an instrumented DynamoDB client, the SDK adds the table name to the segment for calls that target a table. In the console, each table appears as a separate node in the service map, with a generic DynamoDB node for calls that don't target a table.

Example Subsegment for a call to DynamoDB to save an item

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

When you access named resources, calls to the following services create additional nodes in the service map. Calls that don't target specific resources create a generic node for the service.

- **Amazon DynamoDB** – Table name
- **Amazon Simple Storage Service** – Bucket and key name
- **Amazon Simple Queue Service** – Queue name

Tracing calls to downstream HTTP web services with the X-Ray SDK for Go

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

When your application makes calls to microservices or public HTTP APIs, you can use the `xray.Client` to instrument those calls as subsegments of your Go application, as shown in the following example, where *http-client* is an HTTP client.

The client creates a shallow copy of the provided HTTP client, defaulting to `http.DefaultClient`, with roundtripper wrapped with `xray.RoundTripper`.

Example

<caption>**main.go – HTTP client**</caption>

```
myClient := xray.Client(http-client)
```

<caption>**main.go – Trace downstream HTTP call with ctxhttp library**</caption>

The following example instruments the outgoing HTTP call with the ctxhttp library using `xray.Client`. `ctx` can be passed from the upstream call. This ensures that the existing segment context is used. For example, X-Ray does not allow a new segment to be created within a Lambda function, so the existing Lambda segment context should be used.

```
resp, err := ctxhttp.Get(ctx, xray.Client(nil), url)
```

Tracing SQL queries with the X-Ray SDK for Go

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

To trace SQL calls to PostgreSQL or MySQL, replacing `sql.Open` calls to `xray.SQLContext`, as shown in the following example. Use URLs instead of configuration strings if possible.

Example main.go

```
func main() {
  db, err := xray.SQLContext("postgres", "postgres://user:password@host:port/db")
  row, err := db.QueryRowContext(ctx, "SELECT 1") // Use as normal
}
```

Generating custom subsegments with the X-Ray SDK for Go

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

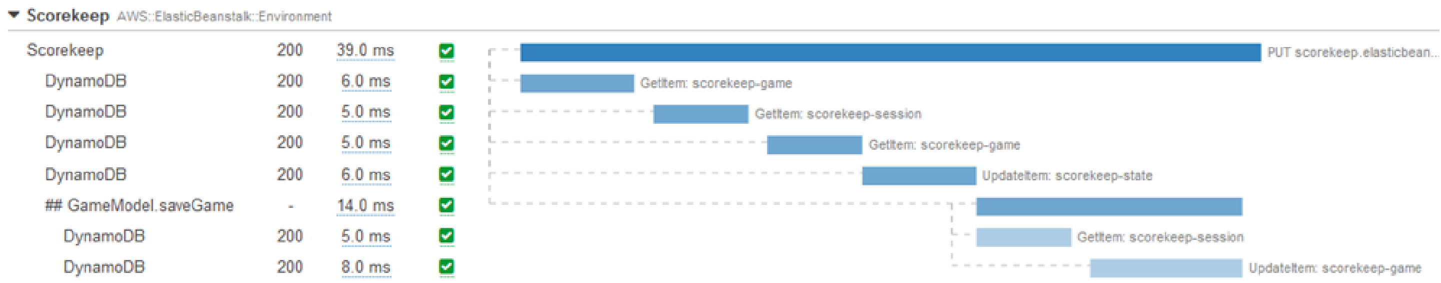
Subsegments extend a trace's segment with details about work done in order to serve a request. Each time you make a call with an instrumented client, the X-Ray SDK records the information generated in a subsegment. You can create additional subsegments to group other subsegments, to measure the performance of a section of code, or to record annotations and metadata.

Use the `Capture` method to create a subsegment around a function.

Example main.go – Custom subsegment

```
func criticalSection(ctx context.Context) {
  //this is an example of a subsegment
  xray.Capture(ctx, "GameModel.saveGame", func(ctx1 context.Context) error {
    var err error

    section.Lock()
    result := someLockedResource.Go()
    section.Unlock()

    xray.AddMetadata(ctx1, "ResourceResult", result)
  })
```

The following screenshot shows an example of how the `saveGame` subsegment might appear in traces for the application `Scorekeep`.



▼ Scorekeep AWS::ElasticBeanstalk::Environment

Scorekeep	200	39.0 ms	✓	PUT scorekeep.elasticbean...
DynamoDB	200	6.0 ms	✓	GetItem: scorekeep-game
DynamoDB	200	5.0 ms	✓	GetItem: scorekeep-session
DynamoDB	200	5.0 ms	✓	GetItem: scorekeep-game
DynamoDB	200	6.0 ms	✓	UpdateItem: scorekeep-state
## GameModel.saveGame	-	14.0 ms	✓	
DynamoDB	200	5.0 ms	✓	GetItem: scorekeep-session
DynamoDB	200	8.0 ms	✓	UpdateItem: scorekeep-game

Add annotations and metadata to segments with the X-Ray SDK for Go

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can record additional information about requests, the environment, or your application with annotations and metadata. You can add annotations and metadata to the segments that the X-Ray SDK creates, or to custom subsegments that you create.

Annotations are key-value pairs with string, number, or Boolean values. Annotations are indexed for use with filter expressions. Use annotations to record data that you want to use to group traces in the console, or when calling the `GetTraceSummaries` API.

Metadata are key-value pairs that can have values of any type, including objects and lists, but are not indexed for use with filter expressions. Use metadata to record additional data that you want stored in the trace but don't need to use with search.

In addition to annotations and metadata, you can also record user ID strings on segments. User IDs are recorded in a separate field on segments and are indexed for use with search.

Sections

- Recording annotations with the X-Ray SDK for Go
- Recording metadata with the X-Ray SDK for Go
- Recording user IDs with the X-Ray SDK for Go

Recording annotations with the X-Ray SDK for Go

Use annotations to record information on segments that you want indexed for search.

Annotation Requirements

- **Keys –** The key for an X-Ray annotation can have up to 500 alphanumeric characters. You cannot use spaces or symbols other than a dot or period (.)
- **Values –** The value for an X-Ray annotation can have up to 1,000 Unicode characters.
- The number of **Annotations –** You can use up to 50 annotations per trace.

To record annotations, call `AddAnnotation` with a string containing the metadata you want to associate with the segment.

```
xray.AddAnnotation(key string, value interface{})
```

The SDK records annotations as key-value pairs in an `annotations` object in the segment document. Calling `AddAnnotation` twice with the same key overwrites previously recorded values on the same segment.

To find traces that have annotations with specific values, use the `annotation[key]` keyword in a filter expression.

Recording metadata with the X-Ray SDK for Go

Use metadata to record information on segments that you don't need indexed for search.

To record metadata, call `AddMetadata` with a string containing the metadata you want to associate with the segment.

```
xray.AddMetadata(key string, value interface{})
```

Recording user IDs with the X-Ray SDK for Go

Record user IDs on request segments to identify the user who sent the request.

To record user IDs

1. Get a reference to the current segment from `AWSXRay`.

```
import (
  "context"
  "github.com/aws/aws-xray-sdk-go/xray"
)

mySegment := xray.GetSegment(context)
```

2. Call `setUser` with a String ID of the user who sent the request.

```
mySegment.User = "U12345"
```

To find traces for a user ID, use the `user` keyword in a filter expression.

Working with Java

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

There are two ways to instrument your Java application to send traces to X-Ray:

- AWS Distro for OpenTelemetry Java – An AWS distribution that provides a set of open source libraries for sending correlated metrics and traces to multiple AWS monitoring solutions, including Amazon CloudWatch, AWS X-Ray, and Amazon OpenSearch Service, via the AWS Distro for OpenTelemetry Collector.
- AWS X-Ray SDK for Java – A set of libraries for generating and sending traces to X-Ray via the X-Ray daemon.

For more information, see Choosing between the AWS Distro for OpenTelemetry and X-Ray SDKs.

AWS Distro for OpenTelemetry Java

With the AWS Distro for OpenTelemetry (ADOT) Java, you can instrument your applications once and send correlated metrics and traces to multiple AWS monitoring solutions including Amazon CloudWatch, AWS X-Ray, and Amazon OpenSearch Service. Using X-Ray with ADOT requires two components: an *OpenTelemetry SDK* enabled for use with X-Ray, and the *AWS Distro for OpenTelemetry Collector* enabled for use with X-Ray. ADOT Java includes auto-instrumentation support, enabling your application to send traces without code changes.

To get started, see the AWS Distro for OpenTelemetry Java documentation.

For more information about using the AWS Distro for OpenTelemetry with AWS X-Ray and other AWS services, see AWS Distro for OpenTelemetry or the AWS Distro for OpenTelemetry Documentation.

For more information about language support and usage, see AWS Observability on GitHub.

AWS X-Ray SDK for Java

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The X-Ray SDK for Java is a set of libraries for Java web applications that provide classes and methods for generating and sending trace data to the X-Ray daemon. Trace data includes information about incoming HTTP requests served by the application, and calls that the application makes to downstream services using the AWS SDK, HTTP clients, or an SQL database connector. You can also create segments manually and add debug information in annotations and metadata.

The X-Ray SDK for Java is an open source project. You can follow the project and submit issues and pull requests on GitHub: github.com/aws/aws-xray-sdk-java

Start by adding `AWSXRayServletFilter` as a servlet filter to trace incoming requests. A servlet filter creates a segment. While the segment is open, you can use the SDK client's methods to add information to the segment and create subsegments to trace downstream calls. The SDK also automatically records exceptions that your application throws while the segment is open.

Starting in release 1.3, you can instrument your application using aspect-oriented programming (AOP) in Spring. What this means is that you can instrument your application, while it is running on AWS, without adding any code to your application's runtime.

Next, use the X-Ray SDK for Java to instrument your AWS SDK for Java clients by including the SDK Instrumentor submodule in your build configuration. Whenever you make a call to a downstream AWS service or resource with an instrumented client, the SDK records information about the call in a subsegment. AWS services and the resources that you access within the services appear as downstream nodes on the trace map to help you identify errors and throttling issues on individual connections.

If you don't want to instrument all downstream calls to AWS services, you can leave out the Instrumentor submodule and choose which clients to instrument. Instrument individual clients by adding a `TracingHandler` to an AWS SDK service client.

Other X-Ray SDK for Java submodules provide instrumentation for downstream calls to HTTP web APIs and SQL databases. You can use the X-Ray SDK for Java versions of `HTTPClient` and `HTTPClientBuilder` in the Apache HTTP submodule to instrument Apache HTTP clients. To instrument SQL queries, add the SDK's interceptor to your data source.

After you start using the SDK, customize its behavior by configuring the recorder and servlet filter. You can add plugins to record data about the compute resources running your application, customize sampling behavior by defining sampling rules, and set the log level to see more or less information from the SDK in your application logs.

Record additional information about requests and the work that your application does in annotations and metadata. Annotations are simple key-value pairs that are indexed for use with filter expressions, so that you can search for traces that contain specific data. Metadata entries are less restrictive and can record entire objects and arrays — anything that can be serialized into JSON.

Annotations and Metadata

Annotations and metadata are arbitrary text that you add to segments with the X-Ray SDK. Annotations are indexed for use with filter expressions. Metadata are not indexed, but can be viewed in the raw segment with the X-Ray console or API. Anyone that you grant read access to X-Ray can view this data.

When you have a lot of instrumented clients in your code, a single request segment can contain many subsegments, one for each call made with an instrumented client. You can organize and group subsegments by wrapping client calls in custom subsegments. You can create a custom subsegment for an entire function or any section of code, and record metadata and annotations on the subsegment instead of writing everything on the parent segment.

Submodules

You can download the X-Ray SDK for Java from Maven. The X-Ray SDK for Java is split into submodules by use case, with a bill of materials for version management:

- `aws-xray-recorder-sdk-core` (required) – Basic functionality for creating segments and transmitting segments. Includes `AWSXRayServletFilter` for instrumenting incoming requests.
- `aws-xray-recorder-sdk-aws-sdk` – Instruments calls to AWS services made with AWS SDK for Java clients by adding a tracing client as a request handler.
- `aws-xray-recorder-sdk-aws-sdk-v2` – Instruments calls to AWS services made with AWS SDK for Java 2.2 and later clients by adding a tracing client as a request intereceptor.
- `aws-xray-recorder-sdk-aws-sdk-instrumentor` – With `aws-xray-recorder-sdk-aws-sdk`, instruments all AWS SDK for Java clients automatically.
- `aws-xray-recorder-sdk-aws-sdk-v2-instrumentor` – With `aws-xray-recorder-sdk-aws-sdk-v2`, instruments all AWS SDK for Java 2.2 and later clients automatically.
- `aws-xray-recorder-sdk-apache-http` – Instruments outbound HTTP calls made with Apache HTTP clients.
- `aws-xray-recorder-sdk-spring` – Provides interceptors for Spring AOP Framework applications.
- `aws-xray-recorder-sdk-sql-postgres` – Instruments outbound calls to a PostgreSQL database made with JDBC.
- `aws-xray-recorder-sdk-sql-mysql` – Instruments outbound calls to a MySQL database made with JDBC.
- `aws-xray-recorder-sdk-bom` – Provides a bill of materials that you can use to specify the version to use for all submodules.
- `aws-xray-recorder-sdk-metrics` – Publish unsampled Amazon CloudWatch metrics from your collected X-Ray segments.

If you use Maven or Gradle to build your application, add the X-Ray SDK for Java to your build configuration.

For reference documentation of the SDK's classes and methods, see AWS X-Ray SDK for Java API Reference.

Requirements

The X-Ray SDK for Java requires Java 8 or later, Servlet API 3, the AWS SDK, and Jackson.

The SDK depends on the following libraries at compile and runtime:

- AWS SDK for Java version 1.11.398 or later
- Servlet API 3.1.0

These dependencies are declared in the SDK's `pom.xml` file and are included automatically if you build using Maven or Gradle.

If you use a library that is included in the X-Ray SDK for Java, you must use the included version. For example, if you already depend on Jackson at runtime and include JAR files in your deployment for that dependency, you must remove those JAR files because the SDK JAR includes its own versions of Jackson libraries.

Dependency management

The X-Ray SDK for Java is available from Maven:

- **Group** – `com.amazonaws`
- **Artifact** – `aws-xray-recorder-sdk-bom`
- **Version** – `2.11.0`

If you use Maven to build your application, add the SDK as a dependency in your `pom.xml` file.

Example pom.xml - dependencies

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-xray-recorder-sdk-bom</artifactId>
      <version>2.11.0</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-core</artifactId>
  </dependency>
  <dependency>
```

```
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-apache-http</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-aws-sdk</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-aws-sdk-instrumentor</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-sql-postgres</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-sql-mysql</artifactId>
  </dependency>
</dependencies>
```

For Gradle, add the SDK as a compile-time dependency in your `build.gradle` file.

Example build.gradle - dependencies

```
dependencies {
  compile("org.springframework.boot:spring-boot-starter-web")
  testCompile("org.springframework.boot:spring-boot-starter-test")
  compile("com.amazonaws:aws-java-sdk-dynamodb")
  compile("com.amazonaws:aws-xray-recorder-sdk-core")
  compile("com.amazonaws:aws-xray-recorder-sdk-aws-sdk")
  compile("com.amazonaws:aws-xray-recorder-sdk-aws-sdk-instrumentor")
  compile("com.amazonaws:aws-xray-recorder-sdk-apache-http")
  compile("com.amazonaws:aws-xray-recorder-sdk-sql-postgres")
  compile("com.amazonaws:aws-xray-recorder-sdk-sql-mysql")
  testCompile("junit:junit:4.11")
}
dependencyManagement {
    imports {
        mavenBom('com.amazonaws:aws-java-sdk-bom:1.11.39')
        mavenBom('com.amazonaws:aws-xray-recorder-sdk-bom:2.11.0')
    }
}
```

If you use Elastic Beanstalk to deploy your application, you can use Maven or Gradle to build on-instance each time you deploy, instead of building and uploading a large archive that includes all of your dependencies. See the sample application for an example that uses Gradle.

AWS X-Ray auto-instrumentation agent for Java

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The AWS X-Ray auto-instrumentation agent for Java is a tracing solution that instruments your Java web applications with minimal development effort. The agent enables tracing for servlet-based applications and all of the agent's downstream requests made with supported frameworks and libraries. This includes downstream Apache HTTP requests, AWS SDK requests, and SQL queries made using a JDBC driver. The agent propagates X-Ray context, including all active segments and subsegments, across threads. All of the configurations and versatility of the X-Ray SDK are still available with the Java agent. Suitable defaults were chosen to ensure that the agent works with minimal effort.

The X-Ray agent solution is best suited for servlet-based, request-response Java web application servers. If your application uses an asynchronous framework, or is not well modeled as a request-response service, you might want to consider manual instrumentation with the SDK instead.

The X-Ray agent is built using the Distributed Systems Comprehension toolkit, or DiSCo. DiSCo is an open source framework for building Java agents that can be used in distributed systems. While it is not necessary to understand DiSCo to use the X-Ray agent, you can learn more about the project by visiting its homepage on GitHub. The X-Ray agent is also fully open-sourced. To view the source code, make contributions, or raise issues about the agent, visit its repository on GitHub.

Sample application

The eb-java-scorekeep sample application is adapted to be instrumented with the X-Ray agent. This branch contains no servlet filter or recorder configuration, as these functions are done by the agent. To run the application locally or using AWS resources, follow the steps in the sample application's readme file. The instructions for using the sample app to generate X-Ray traces are in the sample app’s tutorial.

Getting started

To get started with the X-Ray auto-instrumentation Java agent in your own application, follow these steps.

1. Run the X-Ray daemon in your environment. For more information, see AWS X-Ray daemon.
2. Download the latest distribution of the agent. Unzip the archive and note its location in your file system. Its contents should look like the following.

   ```
   disco
   ### disco-java-agent.jar
   ### disco-plugins
       ### aws-xray-agent-plugin.jar
       ### disco-java-agent-aws-plugin.jar
       ### disco-java-agent-sql-plugin.jar
       ### disco-java-agent-web-plugin.jar
   ```

3. Modify the JVM arguments of your application to include the following, which enables the agent. Ensure the `-javaagent` argument is placed *before* the `-jar` argument if applicable. The process to modify JVM arguments varies depending on the tools and frameworks you use to launch your Java server. Consult the documentation of your server framework for specific guidance.

   ```
   -javaagent:/<path-to-disco>/disco-java-agent.jar=pluginPath=/<path-to-disco>/disco-
   plugins
   ```

4. To specify how the name of your application appears on the X-Ray console, set the `AWS_XRAY_TRACING_NAME` environment variable or the `com.amazonaws.xray.strategy.tracingName` system property. If no name is provided, a default name is used.
5. Restart your server or container. Incoming requests and their downstream calls are now traced. If you don’t see the expected results, see the section called “Troubleshooting”.

Configuration

The X-Ray agent is configured by an external, user-provided JSON file. By default, this file is at the root of the user's classpath (for example, in their `resources` directory) named `xray-agent.json`. You can configure a custom location for the config file by setting the `com.amazonaws.xray.configFile` system property to the absolute filesystem path of your configuration file.

An example configuration file is shown next.

```
{
    "serviceName": "XRayInstrumentedService",
    "contextMissingStrategy": "LOG_ERROR",
    "daemonAddress": "127.0.0.1:2000",
    "tracingEnabled": true,
    "samplingStrategy": "CENTRAL",
    "traceIdInjectionPrefix": "prefix",
    "samplingRulesManifest": "/path/to/manifest",
    "awsServiceHandlerManifest": "/path/to/manifest",
    "awsSdkVersion": 2,
    "maxStackTraceLength": 50,
    "streamingThreshold": 100,
    "traceIdInjection": true,
    "pluginsEnabled": true,
    "collectSqlQueries": false
}
```

Configuration specification

The following table describes valid values for each property. Property names are case sensitive, but their keys are not. For properties that can be overridden by environment variables and system properties, the order of priority is always environment variable, then system property, and then configuration file. For information about properties that you can override, see Environment variables. All fields are optional.

Property name	Type	Valid values	Description	Environment variable	System property	Default
serviceName	String	Any string	The name of your instrumen	AWS_XRAY_TRACING_NAME	com.amazonaws.xray.strategy	XRayInstrumentedService

Property name	Type	Valid values	Description	Environment variable	System property	Default
			ted service as it will appear in the X-Ray console.		.tracingName	
contextMissingStrategy	String	LOG_ERROR , IGNORE_ERROR	The action taken by the agent when it attempts to use the X-Ray segment context but none is present.	AWS_XRAY_CONTEXT_MISSING	com.amazonaws.xray.strategy.contextMissingStrategy	LOG_ERROR
daemonAddress	String	Formatted IP address and port, or list of TCP and UDP address	The address the agent uses to communicate with the X-Ray daemon.	AWS_XRAY_DAEMON_ADDRESS	com.amazonaws.xray.emitter.daemonAddress	127.0.0.1:2000
tracingEnabled	Boolean	True, False	Enables instrumentation by the X-Ray agent.	AWS_XRAY_TRACING_ENABLED	com.amazonaws.xray.tracingEnabled	TRUE

Property name	Type	Valid values	Description	Environment variable	System property	Default
samplingStrategy	String	CENTRAL, LOCAL, NONE, ALL	The sampling strategy used by the agent. ALL captures all requests, NONE captures no requests. See sampling rules.	N/A	N/A	CENTRAL
traceIdInjectionPrefix	String	Any string	Includes the provided prefix before injected trace IDs in logs.	N/A	N/A	None (empty string)

Property name	Type	Valid values	Description	Environment variable	System property	Default
samplingRulesManifest	String	An absolute file path	The path to a custom sampling rules file to be used as the source of sampling rules for the local sampling strategy, or the fallback rules for the central strategy.	N/A	N/A	DefaultSamplingRules.json
awsServiceHandlerManifest	String	An absolute file path	The path to a custom parameter allow list, which captures additional information from AWS SDK clients.	N/A	N/A	DefaultOperationParameterWhitelist.json

Property name	Type	Valid values	Description	Environment variable	System property	Default
awsSdkVersion	Integer	1, 2	Version of the AWS SDK for Java you're using. Ignored if `awsServiceHandlerManifest` is not also set.	N/A	N/A	2
maxStackTraceLength	Integer	Non-negative integers	The maximum lines of a stack trace to record in a trace.	N/A	N/A	50
streamingThreshold	Integer	Non-negative integers	After at least this many subsegments are closed, they are streamed to the daemon out-of-band to avoid chunks being too large.	N/A	N/A	100

Property name	Type	Valid values	Description	Environment variable	System property	Default
traceIdInjection	Boolean	True, False	Enables X-Ray trace ID injection into logs if the dependencies and configuration described in logging config are also added. Otherwise, does nothing.	N/A	N/A	TRUE
pluginsEnabled	Boolean	True, False	Enables plugins that record metadata about the AWS environments you're operating in. See plugins.	N/A	N/A	TRUE

Property name	Type	Valid values	Description	Environment variable	System property	Default
collectSqlQueries	Boolean	True, False	Records SQL query strings in SQL subsegments on a best-effort basis.	N/A	N/A	FALSE
contextPropagation	Boolean	True, False	Automatically propagates X-Ray context between threads if true. Otherwise, uses Thread Local to store context and manual propagation across threads is required.	N/A	N/A	TRUE

Logging configuration

The X-Ray agent's log level can be configured in the same way as the X-Ray SDK for Java. See Logging for more information on configuring logging with the X-Ray SDK for Java.

Manual instrumentation

If you’d like to perform manual instrumentation in addition to the agent’s auto-instrumentation, add the X-Ray SDK as a dependency to your project. Note that the SDK's custom servlet filters mentioned in Tracing Incoming Requests are not compatible with the X-Ray agent.

> **Note**
>
> You must use the latest version of the X-Ray SDK to perform manual instrumentation while also using the agent.

If you are working in a Maven project, add the following dependencies to your `pom.xml` file.

```
<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-core</artifactId>
    <version>2.11.0</version>
  </dependency>
  </dependencies>
```

If you are working in a Gradle project, add the following dependencies to your `build.gradle` file.

```
implementation 'com.amazonaws:aws-xray-recorder-sdk-core:2.11.0'
```

You can add custom subsegments in addition to annotations, metadata, and user IDs while using the agent, just as you would with the normal SDK. The agent automatically propagates context across threads, so no workarounds to propagate context should be necessary when working with multithreaded applications.

Troubleshooting

Since the agent offers fully automatic instrumentation, it can be difficult to identify the root cause of a problem when you are experiencing issues. If the X-Ray agent is not working as expected for you, review the following problems and solutions. The X-Ray agent and SDK use Jakarta Commons Logging (JCL). To see the logging output, ensure that a bridge connecting JCL to your logging backend is on the classpath, as in the following example: `log4j-jcl` or `jcl-over-slf4j`.

Problem: I’ve enabled the Java agent on my application but don’t see anything on the X-Ray console

Is the X-Ray daemon running on the same machine?

If not, see the X-Ray daemon documentation to set it up.

In your application logs, do you see a message like "Initializing the X-Ray agent recorder"?

If you have correctly added the agent to your application, this message is logged at INFO level when your application starts, before it starts taking requests. If this message is not there, then the Java agent is not running with your Java process. Make sure you’ve followed all the setup steps correctly with no typos.

In your application logs, do you see several error messages saying something like "Suppressing AWS X-Ray context missing exception"?

These errors occur because the agent is trying to instrument downstream requests, like AWS SDK requests or SQL queries, but the agent was unable to automatically create a segment. If you see many of these errors, the agent might not be the best tool for your use case and you might want to consider manual instrumentation with the X-Ray SDK instead. Alternatively, you can enable X-Ray SDK debug logs to see the stack trace of where the context-missing exceptions are occurring. You can wrap these portions of your code with custom segments, which should resolve these errors. For an example of wrapping downstream requests with custom segments, see the sample code in instrumenting startup code.

Problem: Some of the segments I expect do not appear on the X-Ray console

Does your application use multithreading?

If some segments that you expect to be created are not appearing in your console, background threads in your application might be the cause. If your application performs tasks using background threads that are “fire and forget,” like making a one-off call to a Lambda function with the AWS SDK, or polling some HTTP endpoint periodically, that may confuse the agent while it is propagating context across threads. To verify this is your problem, enable X-Ray SDK debug logs and check for messages like: *Not emitting segment named <NAME > as it parents in-progress subsegments*. To work around this, you can try joining the background threads before your server returns to ensure all the work done in them is recorded. Or, you can set the agent’s `contextPropagation` configuration to `false` to disable context propagation in background threads. If you do this, you'll have to manually instrument those threads with custom segments or ignore the context missing exceptions they produce.

Have you set up sampling rules?

If there are seemingly random or unexpected segments appearing on the X-Ray console, or the segments you expect to be on the console aren't, you might be experiencing a sampling issue. The X-Ray agent applies centralized sampling to all segments it creates, using the rules from the X-Ray console. The default rule is 1 segment per second, plus 5% of segments afterward, are sampled. This means segments that are created rapidly with the agent might not be sampled. To resolve this, you should create custom sampling rules on the X-Ray console that appropriately sample the desired segments. For more information, see sampling.

Configuring the X-Ray SDK for Java

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The X-Ray SDK for Java includes a class named `AWSXRay` that provides the global recorder. This is a `TracingHandler` that you can use to instrument your code. You can configure the global recorder to customize the `AWSXRayServletFilter` that creates segments for incoming HTTP calls.

Sections

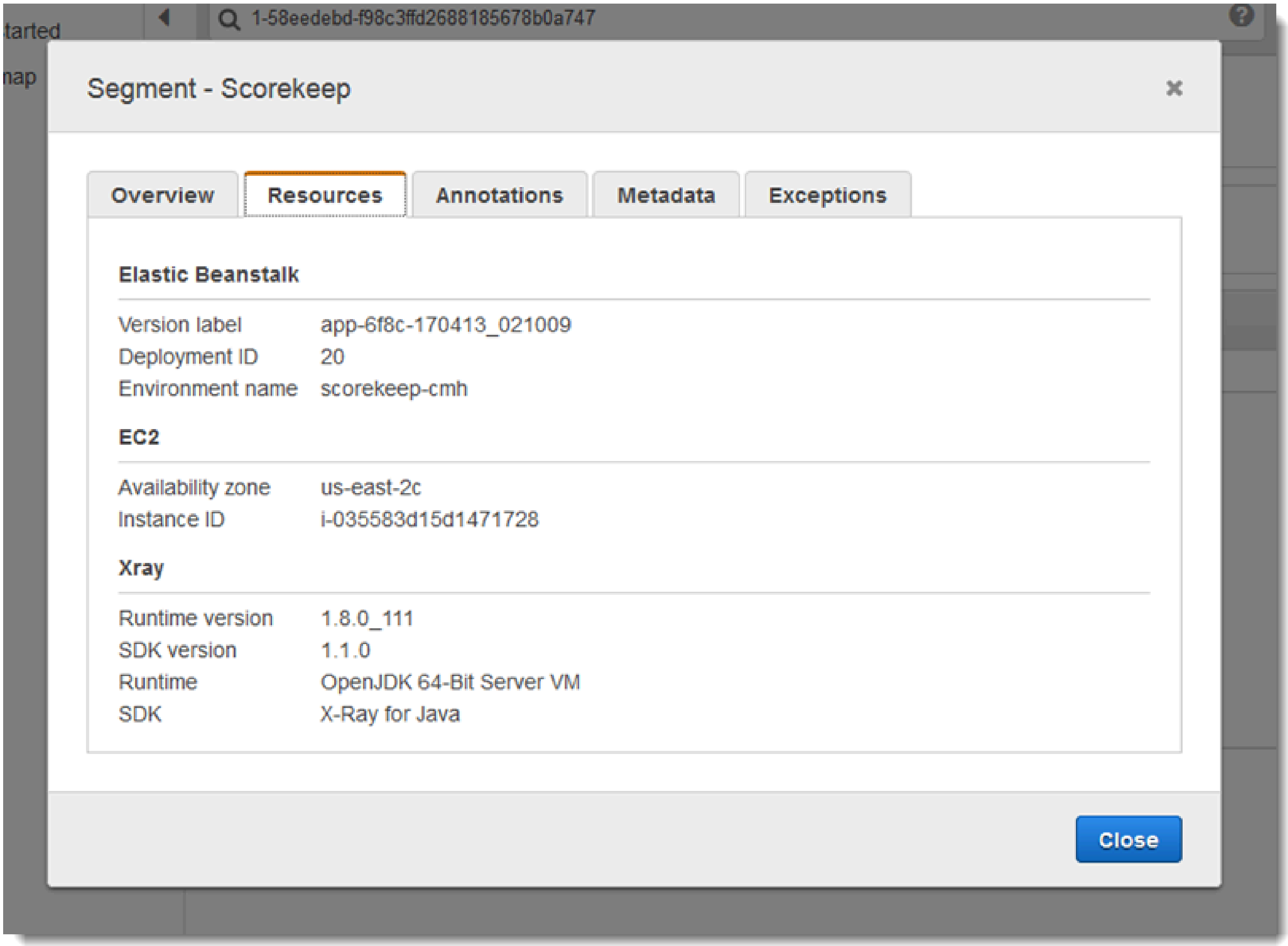
- Service plugins
- Sampling rules
- Logging
- Segment listeners
- Environment variables
- System properties

Service plugins

Use `plugins` to record information about the service hosting your application.

Plugins

- Amazon EC2 – `EC2Plugin` adds the instance ID, Availability Zone, and the CloudWatch Logs Group.
- Elastic Beanstalk – `ElasticBeanstalkPlugin` adds the environment name, version label, and deployment ID.
- Amazon ECS – `ECSPlugin` adds the container ID.
- Amazon EKS – `EKSPlugin` adds the container ID, cluster name, pod ID, and the CloudWatch Logs Group.



Segment - Scorekeep

Elastic Beanstalk

Version label	app-6f8c-170413_021009
Deployment ID	20
Environment name	scorekeep-cmh

EC2

Availability zone	us-east-2c
Instance ID	i-035583d15d1471728

Xray

Runtime version	1.8.0_111
SDK version	1.1.0
Runtime	OpenJDK 64-Bit Server VM
SDK	X-Ray for Java

To use a plugin, call `withPlugin` on your `AWSXRayRecorderBuilder`.

Example src/main/java/scorekeep/WebConfig.java - recorder

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.plugins.ElasticBeanstalkPlugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;

@Configuration
public class WebConfig {
...
  static {
    AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder.standard().withPlugin(new
 EC2Plugin()).withPlugin(new ElasticBeanstalkPlugin());

    URL ruleFile = WebConfig.class.getResource("/sampling-rules.json");
    builder.withSamplingStrategy(new LocalizedSamplingStrategy(ruleFile));

    AWSXRay.setGlobalRecorder(builder.build());
  }
}
```

The SDK also uses plugin settings to set the `origin` field on the segment. This indicates the type of AWS resource that runs your application. When you use multiple plugins, the SDK uses the following resolution order to determine the origin: ElasticBeanstalk > EKS > ECS > EC2.

Sampling rules

The SDK uses the sampling rules you define in the X-Ray console to determine which requests to record. The default rule traces the first request each second, and five percent of any additional requests across all services sending traces to X-Ray. Create additional rules in the X-Ray console to customize the amount of data recorded for each of your applications.

The SDK applies custom rules in the order in which they are defined. If a request matches multiple custom rules, the SDK applies only the first rule.

Note

If the SDK can't reach X-Ray to get sampling rules, it reverts to a default local rule of the first request each second, and five percent of any additional requests per host. This can occur if the host doesn't have permission to call sampling APIs, or can't connect to the X-Ray daemon, which acts as a TCP proxy for API calls made by the SDK.

You can also configure the SDK to load sampling rules from a JSON document. The SDK can use local rules as a backup for cases where X-Ray sampling is unavailable, or use local rules exclusively.

Example sampling-rules.json

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

This example defines one custom rule and a default rule. The custom rule applies a five-percent sampling rate with no minimum number of requests to trace for paths under `/api/move/`. The default rule traces the first request each second and 10 percent of additional requests.

The disadvantage of defining rules locally is that the fixed target is applied by each instance of the recorder independently, instead of being managed by the X-Ray service. As you deploy more hosts, the fixed rate is multiplied, making it harder to control the amount of data recorded.

On AWS Lambda, you cannot modify the sampling rate. If your function is called by an instrumented service, calls that generated requests that were sampled by that service will be recorded by Lambda. If active tracing is enabled and no tracing header is present, Lambda makes the sampling decision.

To provide backup rules in Spring, configure the global recorder with a `CentralizedSamplingStrategy` in a configuration class.

Example src/main/java/myapp/WebConfig.java - recorder configuration

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;

@Configuration
public class WebConfig {

  static {
  AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder.standard().withPlugin(new
 EC2Plugin());

  URL ruleFile = WebConfig.class.getResource("/sampling-rules.json");
  builder.withSamplingStrategy(new CentralizedSamplingStrategy(ruleFile));

  AWSXRay.setGlobalRecorder(builder.build());
}
```

For Tomcat, add a listener that extends `ServletContextListener` and register the listener in the deployment descriptor.

Example src/com/myapp/web/Startup.java

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;

import java.net.URL;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class Startup implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent event) {
        AWSXRayRecorderBuilder builder =
 AWSXRayRecorderBuilder.standard().withPlugin(new EC2Plugin());

        URL ruleFile = Startup.class.getResource("/sampling-rules.json");
        builder.withSamplingStrategy(new CentralizedSamplingStrategy(ruleFile));

        AWSXRay.setGlobalRecorder(builder.build());
    }

    @Override
    public void contextDestroyed(ServletContextEvent event) { }
}
```

Example WEB-INF/web.xml

```
...
  <listener>
    <listener-class>com.myapp.web.Startup</listener-class>
  </listener>
```

To use local rules only, replace the `CentralizedSamplingStrategy` with a `LocalizedSamplingStrategy`.

```
builder.withSamplingStrategy(new LocalizedSamplingStrategy(ruleFile));
```

Logging

By default, the SDK outputs `ERROR`-level messages to your application logs. You can enable debug-level logging on the SDK to output more detailed logs to your application log file. Valid log levels are `DEBUG`, `INFO`, `WARN`, `ERROR`, and `FATAL`. `FATAL` log level silences all log messages because the SDK does not log at fatal level.

Example application.properties

Set the logging level with the `logging.level.com.amazonaws.xray` property.

```
logging.level.com.amazonaws.xray = DEBUG
```

Use debug logs to identify issues, such as unclosed subsegments, when you generate subsegments manually.

Trace ID injection into logs

To expose the current fully qualified trace ID to your log statements, you can inject the ID into the mapped diagnostic context (MDC). Using the `SegmentListener` interface, methods are called from the X-Ray recorder during segment lifecycle events. When a segment or subsegment begins, the qualified trace ID is injected into the MDC with the key `AWS-XRAY-TRACE-ID`. When that segment ends, the key is removed from the MDC. This exposes the trace ID to the logging library in use. When a subsegment ends, its parent ID is injected into the MDC.

Example fully qualified trace ID

The fully qualified ID is represented as `TraceID@EntityID`

```
1-5df42873-011e96598b447dfca814c156@541b3365be3dafc3
```

This feature works with Java applications instrumented with the AWS X-Ray SDK for Java, and supports the following logging configurations:

- SLF4J front-end API with Logback backend
- SLF4J front-end API with Log4J2 backend
- Log4J2 front-end API with Log4J2 backend

See the following tabs for the needs of each front end and each backend.

SLF4J Frontend

1. Add the following Maven dependency to your project.

   ```
   <dependency>
       <groupId>com.amazonaws</groupId>
       <artifactId>aws-xray-recorder-sdk-slf4j</artifactId>
       <version>2.11.0</version>
   </dependency>
   ```

2. Include the `withSegmentListener` method when building the `AWSXRayRecorder`. This adds a `SegmentListener` class, which automatically injects new trace IDs into the SLF4J MDC.

   The `SegmentListener` takes an optional string as a parameter to configure the log statement prefix. The prefix can be configured in the following ways:

- **None** – Uses the default `AWS-XRAY-TRACE-ID` prefix.
- **Empty** – Uses an empty string (e.g. `""`).
- **Custom** – Uses a custom prefix as defined in the string.

Example `AWSXRayRecorderBuilder` statement

```
AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder
        .standard().withSegmentListener(new SLF4JSegmentListener("CUSTOM-
PREFIX"));
```

Log4J2 front end

1. Add the following Maven dependency to your project.

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-log4j</artifactId>
    <version>2.11.0</version>
</dependency>
```

2. Include the `withSegmentListener` method when building the `AWSXRayRecorder`. This will add a `SegmentListener` class, which automatically injects new fully qualified trace IDs into the SLF4J MDC.

   The `SegmentListener` takes an optional string as a parameter to configure the log statement prefix. The prefix can be configured in the following ways:

   - **None** – Uses the default `AWS-XRAY-TRACE-ID` prefix.
   - **Empty** – Uses an empty string (e.g. `""`) and removes the prefix.
   - **Custom** – Uses the custom prefix defined in the string.

Example `AWSXRayRecorderBuilder` statement

```
AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder
        .standard().withSegmentListener(new Log4JSegmentListener("CUSTOM-
PREFIX"));
```

Logback backend

To insert the trace ID into your log events, you must modify the logger's `PatternLayout`, which formats each logging statement.

1. Find where the `patternLayout` is configured. You can do this programmatically, or through an XML configuration file. To learn more, see Logback configuration.
2. Insert `%X{AWS-XRAY-TRACE-ID}` anywhere in the `patternLayout` to insert the trace ID in future logging statements. `%X{}` indicates that you are retrieving a value with the provided key from the MDC. To learn more about PatternLayouts in Logback, see PatternLayout.

Log4J2 backend

1. Find where the `patternLayout` is configured. You can do this programmatically, or through a configuration file written in XML, JSON, YAML, or properties format.

   To learn more about configuring Log4J2 through a configuration file, see Configuration.

   To learn more about configuring Log4J2 programmatically, see Programmatic Configuration.
2. Insert `%X{AWS-XRAY-TRACE-ID}` anywhere in the `PatternLayout` to insert the trace ID in future logging statements. `%X{}` indicates that you are retrieving a value with the provided key from the MDC. To learn more about PatternLayouts in Log4J2, see Pattern Layout.

Trace ID Injection Example

The following shows a `PatternLayout` string modified to include the trace ID. The trace ID is printed after the thread name (`%t`) and before the log level (`%-5p`).

Example `PatternLayout` With ID injection

```
%d{HH:mm:ss.SSS} [%t] %X{AWS-XRAY-TRACE-ID} %-5p %m%n
```

AWS X-Ray automatically prints the key and the trace ID in the log statement for easy parsing. The following shows a log statement using the modified `PatternLayout`.

Example Log statement with ID injection

```
2019-09-10 18:58:30.844 [nio-5000-exec-4]  AWS-XRAY-TRACE-ID:
 1-5d77f256-19f12e4eaa02e3f76c78f46a@1ce7df03252d99e1 WARN 1 - Your logging message
 here
```

The logging message itself is housed in the pattern `%m` and is set when calling the logger.

Segment listeners

Segment listeners are an interface to intercept lifecycle events such as the beginning and ending of segments produced by the `AWSXRayRecorder`. Implementation of a segment listener event function might be to add the same annotation to all subsegments when they are created with `onBeginSubsegment`, log a message after each segment is sent to the daemon using `afterEndSegment`, or to record queries sent by the SQL interceptors using `beforeEndSubsegment` to verify if the subsegment represents an SQL query, adding additional metadata if so.

To see the full list of `SegmentListener` functions, visit the documentation for the AWS X-Ray Recorder SDK for Java API.

The following example shows how to add a consistent annotation to all subsegments on creation with `onBeginSubsegment` and to print a log message at the end of each segment with `afterEndSegment`.

Example MySegmentListener.java

```
import com.amazonaws.xray.entities.Segment;
import com.amazonaws.xray.entities.Subsegment;
import com.amazonaws.xray.listeners.SegmentListener;

public class MySegmentListener implements SegmentListener {
    .....

    @Override
    public void onBeginSubsegment(Subsegment subsegment) {
        subsegment.putAnnotation("annotationKey", "annotationValue");
    }

    @Override
    public void afterEndSegment(Segment segment) {
        // Be mindful not to mutate the segment
        logger.info("Segment with ID " + segment.getId());
    }
}
```

This custom segment listener is then referenced when building the `AWSXRayRecorder`.

Example AWSXRayRecorderBuilder statement

```
AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder
        .standard().withSegmentListener(new MySegmentListener());
```

Environment variables

You can use environment variables to configure the X-Ray SDK for Java. The SDK supports the following variables.

- `AWS_XRAY_CONTEXT_MISSING` – Set to `RUNTIME_ERROR` to throw exceptions when your instrumented code attempts to record data when no segment is open.

  Valid Values

  - `RUNTIME_ERROR` – Throw a runtime exception.
  - `LOG_ERROR` – Log an error and continue (default).
  - `IGNORE_ERROR` – Ignore error and continue.

  Errors related to missing segments or subsegments can occur when you attempt to use an instrumented client in startup code that runs when no request is open, or in code that spawns a new thread.

- `AWS_XRAY_DAEMON_ADDRESS` – Set the host and port of the X-Ray daemon listener. By default, the SDK uses `127.0.0.1:2000` for both trace data (UDP) and sampling (TCP). Use this variable if you have configured the daemon to listen on a different port or if it is running on a different host.

  Format

  - **Same port** – `*address*:*port*`
  - **Different ports** – `tcp:*address*:*port* udp:*address*:*port*`

- `AWS_LOG_GROUP` – Set the name of a log group to log group associated with your application. If your log group uses the same AWS account and region as your application, X-Ray will automatically search for your application's segment data using this specified log group. For more information about log groups, see Working with log groups and streams.

- `AWS_XRAY_TRACING_NAME` – Set a service name that the SDK uses for segments. Overrides the service name that you set on the servlet filter's segment naming strategy.

Environment variables override equivalent system properties and values set in code.

System properties

You can use system properties as a JVM-specific alternative to environment variables. The SDK supports the following properties:

- `com.amazonaws.xray.strategy.tracingName` – Equivalent to `AWS_XRAY_TRACING_NAME`.
- `com.amazonaws.xray.emitters.daemonAddress` – Equivalent to `AWS_XRAY_DAEMON_ADDRESS`.
- `com.amazonaws.xray.strategy.contextMissingStrategy` – Equivalent to `AWS_XRAY_CONTEXT_MISSING`.

If both a system property and the equivalent environment variable are set, the environment variable value is used. Either method overrides values set in code.

Tracing incoming requests with the X-Ray SDK for Java

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can use the X-Ray SDK to trace incoming HTTP requests that your application serves on an EC2 instance in Amazon EC2, AWS Elastic Beanstalk, or Amazon ECS.

Use a `Filter` to instrument incoming HTTP requests. When you add the X-Ray servlet filter to your application, the X-Ray SDK for Java creates a segment for each sampled request. This segment includes timing, method, and disposition of the HTTP request. Additional instrumentation creates subsegments on this segment.

> **Note**
>
> For AWS Lambda functions, Lambda creates a segment for each sampled request. See AWS Lambda and AWS X-Ray for more information.

Each segment has a name that identifies your application in the service map. The segment can be named statically, or you can configure the SDK to name it dynamically based on the host header in the incoming request. Dynamic naming lets you group traces based on the domain name in the request, and apply a default name if the name doesn't match an expected pattern (for example, if the host header is forged).

> **Forwarded Requests**
>
> If a load balancer or other intermediary forwards a request to your application, X-Ray takes the client IP from the `X-Forwarded-For` header in the request instead of from the source IP in the IP packet. The client IP that is recorded for a forwarded request can be forged, so it should not be trusted.

When a request is forwarded, the SDK sets an additional field in the segment to indicate this. If the segment contains the field `x_forwarded_for` set to `true`, the client IP was taken from the `X-Forwarded-For` header in the HTTP request.

The message handler creates a segment for each incoming request with an `http` block that contains the following information:

- **HTTP method** – GET, POST, PUT, DELETE, etc.
- **Client address** – The IP address of the client that sent the request.
- **Response code** – The HTTP response code for the completed request.
- **Timing** – The start time (when the request was received) and end time (when the response was sent).

- **User agent** — The `user-agent` from the request.
- **Content length** — The `content-length` from the response.

Sections

- Adding a tracing filter to your application (Tomcat)
- Adding a tracing filter to your application (spring)
- Configuring a segment naming strategy

Adding a tracing filter to your application (Tomcat)

For Tomcat, add a `<filter>` to your project's `web.xml` file. Use the `fixedName` parameter to specify a service name to apply to segments created for incoming requests.

Example WEB-INF/web.xml - Tomcat

```
<filter>
  <filter-name>AWSXRayServletFilter</filter-name>
  <filter-class>com.amazonaws.xray.javax.servlet.AWSXRayServletFilter</filter-class>
  <init-param>
    <param-name>fixedName</param-name>
    <param-value>MyApp</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>AWSXRayServletFilter</filter-name>
  <url-pattern>*</url-pattern>
</filter-mapping>
```

Adding a tracing filter to your application (spring)

For Spring, add a `Filter` to your `WebConfig` class. Pass the segment name to the `AWSXRayServletFilter` constructor as a string.

Example src/main/java/myapp/WebConfig.java - spring

```
package myapp;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Bean;
import javax.servlet.Filter;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;

@Configuration
public class WebConfig {

  @Bean
  public Filter TracingFilter() {
    return new AWSXRayServletFilter("Scorekeep");
  }
}
```

Configuring a segment naming strategy

AWS X-Ray uses a *service name* to identify your application and distinguish it from the other applications, databases, external APIs, and AWS resources that your application uses. When the X-Ray SDK generates segments for incoming requests, it records your application's service name in the segment's name field.

The X-Ray SDK can name segments after the hostname in the HTTP request header. However, this header can be forged, which could result in unexpected nodes in your service map. To prevent the SDK from naming segments incorrectly due to requests with forged host headers, you must specify a default name for incoming requests.

If your application serves requests for multiple domains, you can configure the SDK to use a dynamic naming strategy to reflect this in segment names. A dynamic naming strategy allows the SDK to use the hostname for requests that match an expected pattern, and apply the default name to requests that don't.

For example, you might have a single application serving requests to three subdomains– `www.example.com`, `api.example.com`, and `static.example.com`. You can use a dynamic naming strategy with the pattern `*.example.com` to identify segments for each subdomain with a different name, resulting in three service nodes on the service map. If your application receives requests with a hostname that doesn't match the pattern, you will see a fourth node on the service map with a fallback name that you specify.

To use the same name for all request segments, specify the name of your application when you initialize the servlet filter, as shown in the previous section. This has the same effect as creating a fixed SegmentNamingStrategy by calling `SegmentNamingStrategy.fixed()` and passing it to the `AWSXRayServletFilter` constructor.

Note

You can override the default service name that you define in code with the AWS_XRAY_TRACING_NAME environment variable.

A dynamic naming strategy defines a pattern that hostnames should match, and a default name to use if the hostname in the HTTP request does not match the pattern. To name segments dynamically in Tomcat, use the dynamicNamingRecognizedHosts and dynamicNamingFallbackName to define the pattern and default name, respectively.

Example WEB-INF/web.xml - servlet filter with dynamic naming

```
<filter>
  <filter-name>AWSXRayServletFilter</filter-name>
  <filter-class>com.amazonaws.xray.javax.servlet.AWSXRayServletFilter</filter-class>
  <init-param>
    <param-name>dynamicNamingRecognizedHosts</param-name>
    <param-value>*.example.com</param-value>
  </init-param>
  <init-param>
    <param-name>dynamicNamingFallbackName</param-name>
    <param-value>MyApp</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>AWSXRayServletFilter</filter-name>
  <url-pattern>*</url-pattern>
</filter-mapping>
```

For Spring, create a dynamic SegmentNamingStrategy by calling SegmentNamingStrategy.dynamic(), and pass it to the AWSXRayServletFilter constructor.

Example src/main/java/myapp/WebConfig.java - servlet filter with dynamic naming

```
package myapp;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Bean;
import javax.servlet.Filter;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;
import com.amazonaws.xray.strategy.SegmentNamingStrategy;
```

```
@Configuration
public class WebConfig {

  @Bean
  public Filter TracingFilter() {
    return new AWSXRayServletFilter(SegmentNamingStrategy.dynamic("MyApp",
 "*.example.com"));
  }
}
```

Tracing AWS SDK calls with the X-Ray SDK for Java

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

When your application makes calls to AWS services to store data, write to a queue, or send notifications, the X-Ray SDK for Java tracks the calls downstream in subsegments. Traced AWS services and resources that you access within those services (for example, an Amazon S3 bucket or Amazon SQS queue), appear as downstream nodes on the trace map in the X-Ray console.

The X-Ray SDK for Java automatically instruments all AWS SDK clients when you include the `aws-sdk` and an `aws-sdk-instrumentor` submodules in your build. If you don't include the Instrumentor submodule, you can choose to instrument some clients while excluding others.

To instrument individual clients, remove the `aws-sdk-instrumentor` submodule from your build and add an `XRayClient` as a `TracingHandler` on your AWS SDK client using the service's client builder.

For example, to instrument an `AmazonDynamoDB` client, pass a tracing handler to `AmazonDynamoDBClientBuilder`.

Example MyModel.java - DynamoDB client

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.handlers.TracingHandler;

...
public class MyModel {
  private AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
        .withRegion(Regions.fromName(System.getenv("AWS_REGION")))
        .withRequestHandlers(new TracingHandler(AWSXRay.getGlobalRecorder()))
        .build();
...
```

For all services, you can see the name of the API called in the X-Ray console. For a subset of services, the X-Ray SDK adds information to the segment to provide more granularity in the service map.

For example, when you make a call with an instrumented DynamoDB client, the SDK adds the table name to the segment for calls that target a table. In the console, each table appears as a separate node in the service map, with a generic DynamoDB node for calls that don't target a table.

Example Subsegment for a call to DynamoDB to save an item

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

When you access named resources, calls to the following services create additional nodes in the service map. Calls that don't target specific resources create a generic node for the service.

- **Amazon DynamoDB** – Table name
- **Amazon Simple Storage Service** – Bucket and key name
- **Amazon Simple Queue Service** – Queue name

To instrument downstream calls to AWS services with AWS SDK for Java 2.2 and later, you can omit the `aws-xray-recorder-sdk-aws-sdk-v2-instrumentor` module from your build configuration. Include the `aws-xray-recorder-sdk-aws-sdk-v2 module` instead, then instrument individual clients by configuring them with a `TracingInterceptor`.

Example AWS SDK for Java 2.2 and later - tracing interceptor

```
import com.amazonaws.xray.interceptors.TracingInterceptor;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
//...
public class MyModel {
private DynamoDbClient client = DynamoDbClient.builder()
.region(Region.US_WEST_2)
.overrideConfiguration(ClientOverrideConfiguration.builder()
.addExecutionInterceptor(new TracingInterceptor())
.build()
)
.build();
//...
```

Tracing calls to downstream HTTP web services with the X-Ray SDK for Java

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more

information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

When your application makes calls to microservices or public HTTP APIs, you can use the X-Ray SDK for Java's version of `HttpClient` to instrument those calls and add the API to the service graph as a downstream service.

The X-Ray SDK for Java includes `DefaultHttpClient` and `HttpClientBuilder` classes that can be used in place of the Apache HttpComponents equivalents to instrument outgoing HTTP calls.

- `com.amazonaws.xray.proxies.apache.http.DefaultHttpClient` - `org.apache.http.impl.client.DefaultHttpClient`
- `com.amazonaws.xray.proxies.apache.http.HttpClientBuilder` - `org.apache.http.impl.client.HttpClientBuilder`

These libraries are in the `aws-xray-recorder-sdk-apache-http` submodule.

You can replace your existing import statements with the X-Ray equivalent to instrument all clients, or use the fully qualified name when you initialize a client to instrument specific clients.

Example HttpClientBuilder

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.http.HttpEntity;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.util.EntityUtils;
import com.amazonaws.xray.proxies.apache.http.HttpClientBuilder;
...
  public String randomName() throws IOException {
    CloseableHttpClient httpclient = HttpClientBuilder.create().build();
    HttpGet httpGet = new HttpGet("http://names.example.com/api/");
    CloseableHttpResponse response = httpclient.execute(httpGet);
    try {
      HttpEntity entity = response.getEntity();
      InputStream inputStream = entity.getContent();
      ObjectMapper mapper = new ObjectMapper();
      Map<String, String> jsonMap = mapper.readValue(inputStream, Map.class);
      String name = jsonMap.get("name");
      EntityUtils.consume(entity);
      return name;
    } finally {
      response.close();
    }
  }
```

When you instrument a call to a downstream web api, the X-Ray SDK for Java records a subsegment with information about the HTTP request and response. X-Ray uses the subsegment to generate an inferred segment for the remote API.

Example Subsegment for a downstream HTTP call

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

Example Inferred segment for a downstream HTTP call

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
  "trace_id": "1-62be1272-1b71c4274f39f122afa64eab",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  },
  "inferred": true
}
```

Tracing SQL queries with the X-Ray SDK for Java

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

SQL Interceptors

Instrument SQL database queries by adding the X-Ray SDK for Java JDBC interceptor to your data source configuration.

- **PostgreSQL** – `com.amazonaws.xray.sql.postgres.TracingInterceptor`
- **MySQL** – `com.amazonaws.xray.sql.mysql.TracingInterceptor`

These interceptors are in the `aws-xray-recorder-sql-postgres` and `aws-xray-recorder-sql-mysql` submodules, respectively. They implement `org.apache.tomcat.jdbc.pool.JdbcInterceptor` and are compatible with Tomcat connection pools.

Note

SQL interceptors do not record the SQL query itself within subsegments for security purposes.

For Spring, add the interceptor in a properties file and build the data source with Spring Boot's `DataSourceBuilder`.

Example `src/main/java/resources/application.properties` - PostgreSQL JDBC interceptor

```
spring.datasource.continue-on-error=true
spring.jpa.show-sql=false
spring.jpa.hibernate.ddl-auto=create-drop
spring.datasource.jdbc-interceptors=com.amazonaws.xray.sql.postgres.TracingInterceptor
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQL94Dialect
```

Example `src/main/java/myapp/WebConfig.java` - Data source

```
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.jdbc.DataSourceBuilder;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

import javax.servlet.Filter;
import javax.sql.DataSource;
import java.net.URL;

@Configuration
@EnableAutoConfiguration
@EnableJpaRepositories("myapp")
public class RdsWebConfig {

  @Bean
  @ConfigurationProperties(prefix = "spring.datasource")
  public DataSource dataSource() {
      logger.info("Initializing PostgreSQL datasource");
      return DataSourceBuilder.create()
              .driverClassName("org.postgresql.Driver")
```

```
                .url("jdbc:postgresql://" + System.getenv("RDS_HOSTNAME") + ":" +
 System.getenv("RDS_PORT") + "/ebdb")
                .username(System.getenv("RDS_USERNAME"))
                .password(System.getenv("RDS_PASSWORD"))
                .build();
  }
...
}
```

For Tomcat, call `setJdbcInterceptors` on the JDBC data source with a reference to the X-Ray SDK for Java class.

Example `src/main/myapp/model.java` - Data source

```
import org.apache.tomcat.jdbc.pool.DataSource;
...
DataSource source = new DataSource();
source.setUrl(url);
source.setUsername(user);
source.setPassword(password);
source.setDriverClassName("com.mysql.jdbc.Driver");
source.setJdbcInterceptors("com.amazonaws.xray.sql.mysql.TracingInterceptor;");
```

The Tomcat JDBC Data Source library is included in the X-Ray SDK for Java, but you can declare it as a provided dependency to document that you use it.

Example `pom.xml` - JDBC data source

```
<dependency>
  <groupId>org.apache.tomcat</groupId>
  <artifactId>tomcat-jdbc</artifactId>
  <version>8.0.36</version>
  <scope>provided</scope>
</dependency>
```

Native SQL Tracing Decorator

- Add `aws-xray-recorder-sdk-sql` to your dependencies.
- Decorate your database datasource, connection, or statement.

```
dataSource = TracingDataSource.decorate(dataSource)
connection = TracingConnection.decorate(connection)
```

```
statement = TracingStatement.decorateStatement(statement)
preparedStatement = TracingStatement.decoratePreparedStatement(preparedStatement,
 sql)
callableStatement = TracingStatement.decorateCallableStatement(callableStatement,
 sql)
```

Generating custom subsegments with the X-Ray SDK for Java

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

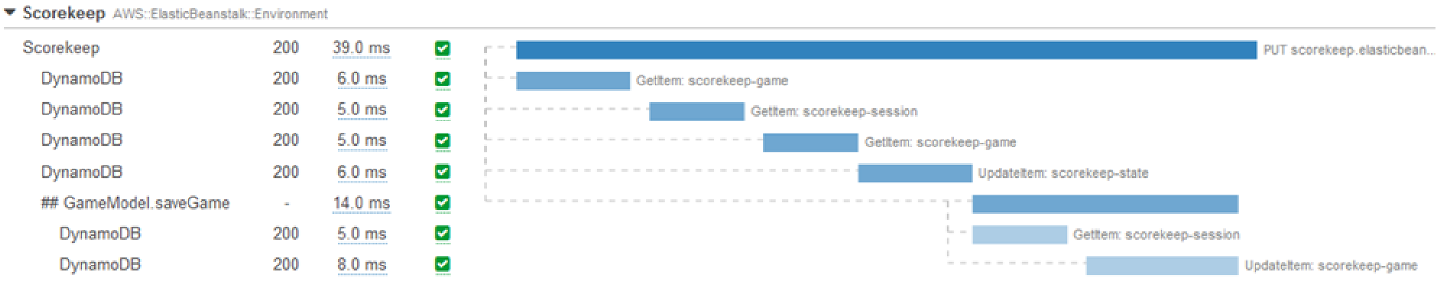
Subsegments extend a trace's segment with details about work done in order to serve a request. Each time you make a call with an instrumented client, the X-Ray SDK records the information generated in a subsegment. You can create additional subsegments to group other subsegments, to measure the performance of a section of code, or to record annotations and metadata.

To manage subsegments, use the `beginSubsegment` and `endSubsegment` methods.

Example GameModel.java - custom subsegment

```
import com.amazonaws.xray.AWSXRay;
...
  public void saveGame(Game game) throws SessionNotFoundException {
    // wrap in subsegment
    Subsegment subsegment = AWSXRay.beginSubsegment("Save Game");
    try {
      // check session
      String sessionId = game.getSession();
      if (sessionModel.loadSession(sessionId) == null ) {
        throw new SessionNotFoundException(sessionId);
      }
      mapper.save(game);
    } catch (Exception e) {
      subsegment.addException(e);
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
  }
```

In this example, the code within the subsegment loads the game's session from DynamoDB with a method on the session model, and uses the AWS SDK for Java's DynamoDB mapper to save the game. Wrapping this code in a subsegment makes the calls DynamoDB children of the `Save Game` subsegment in the trace view in the console.



▼ **Scorekeep** AWS::ElasticBeanstalk::Environment

Scorekeep	200	39.0 ms	✓	PUT scorekeep.elasticbean...
DynamoDB	200	6.0 ms	✓	GetItem: scorekeep-game
DynamoDB	200	5.0 ms	✓	GetItem: scorekeep-session
DynamoDB	200	5.0 ms	✓	GetItem: scorekeep-game
DynamoDB	200	6.0 ms	✓	UpdateItem: scorekeep-state
## GameModel.saveGame	-	14.0 ms	✓	
DynamoDB	200	5.0 ms	✓	GetItem: scorekeep-session
DynamoDB	200	8.0 ms	✓	UpdateItem: scorekeep-game

If the code in your subsegment throws checked exceptions, wrap it in a `try` block and call `AWSXRay.endSubsegment()` in a `finally` block to ensure that the subsegment is always closed. If a subsegment is not closed, the parent segment cannot be completed and won't be sent to X-Ray.

For code that doesn't throw checked exceptions, you can pass the code to `AWSXRay.CreateSubsegment` as a Lambda function.

Example Subsegment Lambda function

```
import com.amazonaws.xray.AWSXRay;

AWSXRay.createSubsegment("getMovies", (subsegment) -> {
    // function code
});
```

When you create a subsegment within a segment or another subsegment, the X-Ray SDK for Java generates an ID for it and records the start time and end time.

Example Subsegment with metadata

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
```

For asynchronous and multi-threaded programming, you must manually pass the subsegment to the `endSubsegment()` method to ensure it is closed correctly because the X-Ray context may be modified during async execution. If an asynchronous subsegment is closed after its parent segment is closed, this method will automatically stream the entire segment to the X-Ray daemon.

Example Asynchronous Subsegment

```
@GetMapping("/api")
public ResponseEntity<?> api() {
  CompletableFuture.runAsync(() -> {
      Subsegment subsegment = AWSXRay.beginSubsegment("Async Work");
      try {
          Thread.sleep(3000);
      } catch (InterruptedException e) {
          subsegment.addException(e);
          throw e;
      } finally {
          AWSXRay.endSubsegment(subsegment);
      }
  });
  return ResponseEntity.ok().build();
}
```

Add annotations and metadata to segments with the X-Ray SDK for Java

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive

updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can record additional information about requests, the environment, or your application with annotations and metadata. You can add annotations and metadata to the segments that the X-Ray SDK creates, or to custom subsegments that you create.

Annotations are key-value pairs with string, number, or Boolean values. Annotations are indexed for use with filter expressions. Use annotations to record data that you want to use to group traces in the console, or when calling the `GetTraceSummaries` API.

Metadata are key-value pairs that can have values of any type, including objects and lists, but are not indexed for use with filter expressions. Use metadata to record additional data that you want stored in the trace but don't need to use with search.

In addition to annotations and metadata, you can also record user ID strings on segments. User IDs are recorded in a separate field on segments and are indexed for use with search.

Sections

- Recording annotations with the X-Ray SDK for Java
- Recording metadata with the X-Ray SDK for Java
- Recording user IDs with the X-Ray SDK for Java

Recording annotations with the X-Ray SDK for Java

Use annotations to record information on segments or subsegments that you want indexed for search.

Annotation Requirements

- **Keys –** The key for an X-Ray annotation can have up to 500 alphanumeric characters. You cannot use spaces or symbols other than a dot or period (.)
- **Values –** The value for an X-Ray annotation can have up to 1,000 Unicode characters.
- The number of **Annotations –** You can use up to 50 annotations per trace.

To record annotations

1. Get a reference to the current segment or subsegment from AWSXRay.

   ```
   import com.amazonaws.xray.AWSXRay;
   import com.amazonaws.xray.entities.Segment;
   ...
   Segment document = AWSXRay.getCurrentSegment();
   ```

   or

   ```
   import com.amazonaws.xray.AWSXRay;
   import com.amazonaws.xray.entities.Subsegment;
   ...
   Subsegment document = AWSXRay.getCurrentSubsegment();
   ```

2. Call putAnnotation with a String key, and a Boolean, Number, or String value.

   ```
   document.putAnnotation("mykey", "my value");
   ```

   The following example shows how to call putAnnotation with a String key that includes a dot, and a Boolean, Number, or String value.

   ```
   document.putAnnotation("testkey.test", "my value");
   ```

The SDK records annotations as key-value pairs in an annotations object in the segment document. Calling putAnnotation twice with the same key overwrites previously recorded values on the same segment or subsegment.

To find traces that have annotations with specific values, use the annotation[*key*] keyword in a filter expression.

Example src/main/java/scorekeep/GameModel.java – Annotations and metadata

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Segment;
import com.amazonaws.xray.entities.Subsegment;
...
  public void saveGame(Game game) throws SessionNotFoundException {
    // wrap in subsegment
```

```
    Subsegment subsegment = AWSXRay.beginSubsegment("## GameModel.saveGame");
    try {
      // check session
      String sessionId = game.getSession();
      if (sessionModel.loadSession(sessionId) == null ) {
        throw new SessionNotFoundException(sessionId);
      }
      Segment segment = AWSXRay.getCurrentSegment();
      subsegment.putMetadata("resources", "game", game);
      segment.putAnnotation("gameid", game.getId());
      mapper.save(game);
    } catch (Exception e) {
      subsegment.addException(e);
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
  }
```

Recording metadata with the X-Ray SDK for Java

Use metadata to record information on segments or subsegments that you don't need indexed for search. Metadata values can be strings, numbers, Booleans, or any object that can be serialized into a JSON object or array.

To record metadata

1. Get a reference to the current segment or subsegment from `AWSXRay`.

   ```
   import com.amazonaws.xray.AWSXRay;
   import com.amazonaws.xray.entities.Segment;
   ...
   Segment document = AWSXRay.getCurrentSegment();
   ```

   or

   ```
   import com.amazonaws.xray.AWSXRay;
   import com.amazonaws.xray.entities.Subsegment;
   ...
   Subsegment document = AWSXRay.getCurrentSubsegment();
   ```

2. Call `putMetadata` with a String namespace, String key, and a Boolean, Number, String, or Object value.

   ```
   document.putMetadata("my namespace", "my key", "my value");
   ```

   or

   Call `putMetadata` with just a key and value.

   ```
   document.putMetadata("my key", "my value");
   ```

If you don't specify a namespace, the SDK uses `default`. Calling `putMetadata` twice with the same key overwrites previously recorded values on the same segment or subsegment.

Example `src/main/java/scorekeep/GameModel.java` – Annotations and metadata

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Segment;
import com.amazonaws.xray.entities.Subsegment;
...
  public void saveGame(Game game) throws SessionNotFoundException {
    // wrap in subsegment
    Subsegment subsegment = AWSXRay.beginSubsegment("## GameModel.saveGame");
    try {
      // check session
      String sessionId = game.getSession();
      if (sessionModel.loadSession(sessionId) == null ) {
        throw new SessionNotFoundException(sessionId);
      }
      Segment segment = AWSXRay.getCurrentSegment();
      subsegment.putMetadata("resources", "game", game);
      segment.putAnnotation("gameid", game.getId());
      mapper.save(game);
    } catch (Exception e) {
      subsegment.addException(e);
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
  }
```

Recording user IDs with the X-Ray SDK for Java

Record user IDs on request segments to identify the user who sent the request.

To record user IDs

1. Get a reference to the current segment from `AWSXRay`.

   ```
   import com.amazonaws.xray.AWSXRay;
   import com.amazonaws.xray.entities.Segment;
   ...
   Segment document = AWSXRay.getCurrentSegment();
   ```

2. Call `setUser` with a string ID of the user who sent the request.

   ```
   document.setUser("U12345");
   ```

You can call `setUser` in your controllers to record the user ID as soon as your application starts processing a request. If you will only use the segment to set the user ID, you can chain the calls in a single line.

Example src/main/java/scorekeep/MoveController.java – User ID

```
import com.amazonaws.xray.AWSXRay;
...
  @RequestMapping(value="/{userId}", method=RequestMethod.POST)
  public Move newMove(@PathVariable String sessionId, @PathVariable String
 gameId, @PathVariable String userId, @RequestBody String move) throws
 SessionNotFoundException, GameNotFoundException, StateNotFoundException,
 RulesException {
    AWSXRay.getCurrentSegment().setUser(userId);
    return moveFactory.newMove(sessionId, gameId, userId, move);
  }
```

To find traces for a user ID, use the `user` keyword in a filter expression.

AWS X-Ray metrics for the X-Ray SDK for Java

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

This topic describes the AWS X-Ray namespace, metrics, and dimensions. You can use the X-Ray SDK for Java to publish unsampled Amazon CloudWatch metrics from your collected X-Ray segments. These metrics are derived from the segment's start and end time, and the error, fault, and throttled status flags. Use these trace metrics to expose retries and dependency issues within subsegments.

CloudWatch is a metrics repository. A metric is the fundamental concept in CloudWatch and represents a time-ordered set of data points. You (or AWS services) publish metrics data points into CloudWatch and you retrieve statistics about those data points as an ordered set of time-series data.

Metrics are uniquely defined by a name, a namespace, and one or more dimensions. Each data point has a timestamp and, optionally, a unit of measure. When you request statistics, the returned data stream is identified by namespace, metric name, and dimension.

For more information about CloudWatch, see the *Amazon CloudWatch User Guide*.

X-Ray CloudWatch metrics

The `ServiceMetrics/SDK` namespace includes the following metrics.

Metric	Statistics available	Description	Units
`Latency`	Average, Minimum, Maximum, Count	The difference between the start and end time.	Milliseconds

Metric	Statistics available	Description	Units
		Average, minimum, and maximum all describe operational latency. Count describes call count.	
`ErrorRate`	Average, Sum	The rate of requests that failed with a `4xx Client Error` status code, resulting in an error.	Percent
`FaultRate`	Average, Sum	The rate of traces that failed with a `5xx Server Error` status code, resulting in a fault.	Percent
`ThrottleRate`	Average, Sum	The rate of throttled traces that return a 429 status code. This is a subset of the `ErrorRate` metric.	Percent
`OkRate`	Average, Sum	The rate of traced requests resulting in an OK status code.	Percent

X-Ray CloudWatch dimensions

Use the dimensions in the following table to refine the metrics returned for your X-Ray instrumented Java applications.

Dimension	Description
`ServiceType`	The type of the service, for example, `AWS::EC2::Instance` or NONE, if not known.
`ServiceName`	The canonical name for the service.

Enable X-Ray CloudWatch metrics

Use the following procedure to enable trace metrics in your instrumented Java application.

To configure trace metrics

1. Add the `aws-xray-recorder-sdk-metrics` package as an Apache Maven dependency. For more information, see X-Ray SDK for Java Submodules.
2. Enable a new `MetricsSegmentListener()` as part of the global recorder build.

Example src/com/myapp/web/Startup.java

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.plugins.ElasticBeanstalkPlugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;

@Configuration
public class WebConfig {
...
  static {
    AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder
                                        .standard()
                                        .withPlugin(new EC2Plugin())
                                        .withPlugin(new ElasticBeanstalkPlugin())
                                        .withSegmentListener(new
 MetricsSegmentListener());

    URL ruleFile = WebConfig.class.getResource("/sampling-rules.json");
    builder.withSamplingStrategy(new LocalizedSamplingStrategy(ruleFile));
    AWSXRay.setGlobalRecorder(builder.build());
  }
}
```

3. Deploy the CloudWatch agent to collect metrics using Amazon Elastic Compute Cloud (Amazon EC2), Amazon Elastic Container Service (Amazon ECS), or Amazon Elastic Kubernetes Service (Amazon EKS):

   - To configure Amazon EC2, see Installing the CloudWatch agent.
   - To configure Amazon ECS, see Monitor Amazon ECS containers using Container Insights.
   - To configure Amazon EKS, see Install the CloudWatch agent by using the Amazon CloudWatch Observability EKS add-on.

4. Configure the SDK to communicate with the CloudWatch agent. By default, the SDK communicates with the CloudWatch agent on the address `127.0.0.1`. You can configure alternate addresses by setting the environment variable or Java property to `address:port`.

   Example Environment variable

   ```
   AWS_XRAY_METRICS_DAEMON_ADDRESS=address:port
   ```

   Example Java property

   ```
   com.amazonaws.xray.metrics.daemonAddress=address:port
   ```

To validate configuration

1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Open the **Metrics** tab to observe the influx of your metrics.
3. (Optional) In the CloudWatch console, on the **Logs** tab, open the `ServiceMetricsSDK` log group. Look for a log stream that matches the host metrics, and confirm the log messages.

Passing segment context between threads in a multithreaded application

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

When you create a new thread in your application, the `AWSXRayRecorder` doesn't maintain a reference to the current segment or subsegment Entity. If you use an instrumented client in the new thread, the SDK tries to write to a segment that doesn't exist, causing a SegmentNotFoundException.

To avoid throwing exceptions during development, you can configure the recorder with a ContextMissingStrategy that tells it to log an error instead. You can configure the strategy in code with SetContextMissingStrategy, or configure equivalent options with an environment variable or system property.

One way to address the error is to use a new segment by calling beginSegment when you start the thread and endSegment when you close it. This works if you are instrumenting code that doesn't run in response to an HTTP request, like code that runs when your application starts.

If you use multiple threads to handle incoming requests, you can pass the current segment or subsegment to the new thread and provide it to the global recorder. This ensures that the information recorded within the new thread is associated with the same segment as the rest of the information recorded about that request. Once the segment is available in the new thread, you can execute any runnable with access to that segment's context using the `segment.run(() -> { ... })` method.

See Using instrumented clients in worker threads for an example.

Using X-Ray with Asynchronous Programming

The X-Ray SDK for Java can be used in asynchronous Java programs with SegmentContextExecutors. The SegmentContextExecutor implements the Executor interface, which means it can be passed into all asynchronous operations of a CompletableFuture. This ensures that any asynchronous operations will be executed with the correct segment in its context.

Example Example App.java: Passing SegmentContextExecutor to CompletableFuture

```
DynamoDbAsyncClient client = DynamoDbAsyncClient.create();

AWSXRay.beginSegment();

// ...

client.getItem(request).thenComposeAsync(response -> {
    // If we did not provide the segment context executor, this request would not be
 traced correctly.
    return client.getItem(request2);
}, SegmentContextExecutors.newSegmentContextExecutor());
```

AOP with Spring and the X-Ray SDK for Java

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

This topic describes how to use the X-Ray SDK and the Spring Framework to instrument your application without changing its core logic. This means that there is now a non-invasive way to instrument your applications running remotely in AWS.

To enable AOP in spring

1. Configure Spring

2. Add a tracing filter to your application
3. Annotate your code or implement an interface
4. Activate X-Ray in your application

Configuring Spring

You can use Maven or Gradle to configure Spring to use AOP to instrument your application.

If you use Maven to build your application, add the following dependency in your `pom.xml` file.

```
<dependency>
     <groupId>com.amazonaws</groupId>
     <artifactId>aws-xray-recorder-sdk-spring</artifactId>
     <version>2.11.0</version>
</dependency>
```

For Gradle, add the following dependency in your `build.gradle` file.

```
compile 'com.amazonaws:aws-xray-recorder-sdk-spring:2.11.0'
```

Configuring Spring Boot

In addition to the Spring dependency described in the previous section, if you're using Spring Boot, add the following dependency if it's not already on your classpath.

Maven:

```
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-aop</artifactId>
     <version>2.5.2</version>
</dependency>
```

Gradle:

```
compile 'org.springframework.boot:spring-boot-starter-aop:2.5.2'
```

Adding a tracing filter to your application

Add a `Filter` to your `WebConfig` class. Pass the segment name to the [`AWSXRayServletFilter`]() constructor as a string. For more information about tracing filters and instrumenting incoming requests, see [Tracing incoming requests with the X-Ray SDK for Java]().

Example src/main/java/myapp/WebConfig.java - spring

```
package myapp;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Bean;
import javax.servlet.Filter;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;

@Configuration
public class WebConfig {

  @Bean
  public Filter TracingFilter() {
    return new AWSXRayServletFilter("Scorekeep");
  }
}
```

Jakarta Support

Spring 6 uses [Jakarta]() instead of Javax for its Enterprise Edition. To support this new namespace, X-Ray has created a parallel set of classes that live in their own Jakarta namespace.

For the filter classes, replace `javax` with `jakarta`. When configuring a segment naming strategy, add `jakarta` before the naming strategy class name, as in the following example:

```
package myapp;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Bean;
import jakarta.servlet.Filter;
import com.amazonaws.xray.jakarta.servlet.AWSXRayServletFilter;
import com.amazonaws.xray.strategy.jakarta.SegmentNamingStrategy;

@Configuration
public class WebConfig {
    @Bean
    public Filter TracingFilter() {
        return new AWSXRayServletFilter(SegmentNamingStrategy.dynamic("Scorekeep"));
    }
}
```

Annotating your code or implementing an interface

Your classes must either be annotated with the `@XRayEnabled` annotation, or implement the `XRayTraced` interface. This tells the AOP system to wrap the functions of the affected class for X-Ray instrumentation.

Activating X-Ray in your application

To activate X-Ray tracing in your application, your code must extend the abstract class `BaseAbstractXRayInterceptor` by overriding the following methods.

- `generateMetadata`—This function allows customization of the metadata attached to the current function's trace. By default, the class name of the executing function is recorded in the metadata. You can add more data if you need additional information.
- `xrayEnabledClasses`—This function is empty, and should remain so. It serves as the host for a pointcut instructing the interceptor about which methods to wrap. Define the pointcut by specifying which of the classes that are annotated with `@XRayEnabled` to trace. The following pointcut statement tells the interceptor to wrap all controller beans annotated with the `@XRayEnabled` annotation.

  ```
  @Pointcut(“@within(com.amazonaws.xray.spring.aop.XRayEnabled) && bean(*Controller)”)
  ```

If your project is using Spring Data JPA, consider extending from `AbstractXRayInterceptor` instead of `BaseAbstractXRayInterceptor`.

Example

The following code extends the abstract class `BaseAbstractXRayInterceptor`.

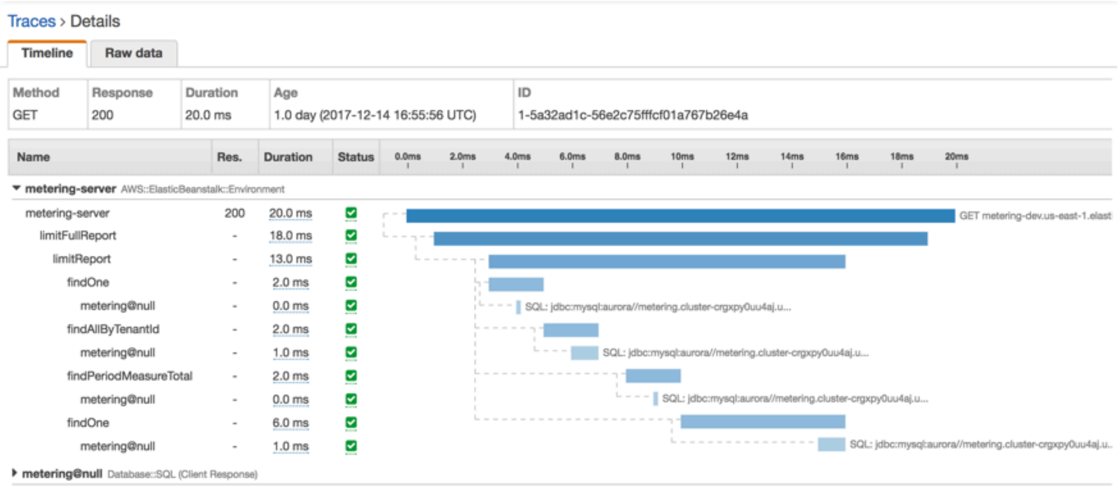
```
@Aspect
@Component
public class XRayInspector extends BaseAbstractXRayInterceptor {
    @Override
```

```
    protected Map<String, Map<String, Object>> generateMetadata(ProceedingJoinPoint
 proceedingJoinPoint, Subsegment subsegment) throws Exception {
        return super.generateMetadata(proceedingJoinPoint, subsegment);
    }

  @Override
  @Pointcut("@within(com.amazonaws.xray.spring.aop.XRayEnabled) && bean(*Controller)")

  public void xrayEnabledClasses() {}

}
```

The following code is a class that will be instrumented by X-Ray.

```
@Service
@XRayEnabled
public class MyServiceImpl implements MyService {
    private final MyEntityRepository myEntityRepository;

    @Autowired
    public MyServiceImpl(MyEntityRepository myEntityRepository) {
        this.myEntityRepository = myEntityRepository;
    }

    @Transactional(readOnly = true)
    public List<MyEntity> getMyEntities(){
        try(Stream<MyEntity> entityStream = this.myEntityRepository.streamAll()){

            return entityStream.sorted().collect(Collectors.toList());
        }
    }
}
```

If you've configured your application correctly, you should see the complete call stack of the application, from the controller down through the service calls, as shown in the following screen shot of the console.



Method	Response	Duration	Age	ID
GET	200	20.0 ms	1.0 day (2017-12-14 16:55:56 UTC)	1-5a32ad1c-56e2c75fffcf01a767b26e4a

Name	Res.	Duration	Status	
▾ **metering-server** AWS::ElasticBeanstalk::Environment				
metering-server	200	20.0 ms	☑	GET metering-dev.us-east-1.elast
limitFullReport	-	18.0 ms	☑	
limitReport	-	13.0 ms	☑	
findOne	-	2.0 ms	☑	
metering@null	-	0.0 ms	☑	SQL: jdbc:mysql:aurora//metering.cluster-crgxpy0uu4aj.u...
findAllByTenantId	-	2.0 ms	☑	
metering@null	-	1.0 ms	☑	SQL: jdbc:mysql:aurora//metering.cluster-crgxpy0uu4aj.u...
findPeriodMeasureTotal	-	2.0 ms	☑	
metering@null	-	0.0 ms	☑	SQL: jdbc:mysql:aurora//metering.cluster-crgxpy0uu4aj.u...
findOne	-	6.0 ms	☑	
metering@null	-	1.0 ms	☑	SQL: jdbc:mysql:aurora//metering.cluster-crgxpy0uu4aj.u..
▸ **metering@null** Database::SQL (Client Response)				

Working with Node.js

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

There are two ways to instrument your Node.js application to send traces to X-Ray:

- AWS Distro for OpenTelemetry JavaScript – An AWS distribution that provides a set of open source libraries for sending correlated metrics and traces to multiple AWS monitoring solutions, including Amazon CloudWatch, AWS X-Ray, and Amazon OpenSearch Service, via the AWS Distro for OpenTelemetry Collector.
- AWS X-Ray SDK for Node.js – A set of libraries for generating and sending traces to X-Ray via the X-Ray daemon.

For more information, see Choosing between the AWS Distro for OpenTelemetry and X-Ray SDKs.

AWS Distro for OpenTelemetry JavaScript

With the AWS Distro for OpenTelemetry (ADOT) JavaScript, you can instrument your applications once and send correlated metrics and traces to multiple AWS monitoring solutions including Amazon CloudWatch, AWS X-Ray, and Amazon OpenSearch Service. Using X-Ray with AWS Distro for OpenTelemetry requires two components: an *OpenTelemetry SDK* enabled for use with X-Ray, and the *AWS Distro for OpenTelemetry Collector* enabled for use with X-Ray.

To get started, see the AWS Distro for OpenTelemetry JavaScript documentation.

Note

ADOT JavaScript is supported for all server-side Node.js applications. ADOT JavaScript is not able to export data to X-Ray from browser clients.

For more information about using the AWS Distro for OpenTelemetry with AWS X-Ray and other AWS services, see AWS Distro for OpenTelemetry or the AWS Distro for OpenTelemetry Documentation.

For more information about language support and usage, see AWS Observability on GitHub.

AWS X-Ray SDK for Node.js

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The X-Ray SDK for Node.js is a library for Express web applications and Node.js Lambda functions that provides classes and methods for generating and sending trace data to the X-Ray daemon. Trace data includes information about incoming HTTP requests served by the application, and calls that the application makes to downstream services using the AWS SDK or HTTP clients.

Note

The X-Ray SDK for Node.js is an open source project that is supported for Node.js versions 14.x and up. You can follow the project and submit issues and pull requests on GitHub: github.com/aws/aws-xray-sdk-node

If you use Express, start by adding the SDK as middleware on your application server to trace incoming requests. The middleware creates a segment for each traced request, and completes the segment when the response is sent. While the segment is open you can use the SDK client's methods to add information to the segment and create subsegments to trace downstream calls. The SDK also automatically records exceptions that your application throws while the segment is open.

For Lambda functions called by an instrumented application or service, Lambda reads the tracing header and traces sampled requests automatically. For other functions, you can configure Lambda to sample and trace incoming requests. In either case, Lambda creates the segment and provides it to the X-Ray SDK.

Note

On Lambda, the X-Ray SDK is optional. If you don't use it in your function, your service map will still include a node for the Lambda service, and one for each Lambda function. By adding the SDK, you can instrument your function code to add subsegments to the function segment recorded by Lambda. See AWS Lambda and AWS X-Ray for more information.

Next, use the X-Ray SDK for Node.js to instrument your AWS SDK for JavaScript in Node.js clients. Whenever you make a call to a downstream AWS service or resource with an instrumented client, the SDK records information about the call in a subsegment. AWS services and the resources that you access within the services appear as downstream nodes on the trace map to help you identify errors and throttling issues on individual connections.

The X-Ray SDK for Node.js also provides instrumentation for downstream calls to HTTP web APIs and SQL queries. Wrap your HTTP client in the SDK's capture method to record information about outgoing HTTP calls. For SQL clients, use the capture method for your database type.

The middleware applies sampling rules to incoming requests to determine which requests to trace. You can configure the X-Ray SDK for Node.js to adjust the sampling behavior or to record information about the AWS compute resources on which your application runs.

Record additional information about requests and the work that your application does in annotations and metadata. Annotations are simple key-value pairs that are indexed for use with filter expressions, so that you can search for traces that contain specific data. Metadata entries are less restrictive and can record entire objects and arrays — anything that can be serialized into JSON.

Annotations and Metadata

Annotations and metadata are arbitrary text that you add to segments with the X-Ray SDK. Annotations are indexed for use with filter expressions. Metadata are not indexed, but can be viewed in the raw segment with the X-Ray console or API. Anyone that you grant read access to X-Ray can view this data.

When you have a lot of instrumented clients in your code, a single request segment can contain a large number of subsegments, one for each call made with an instrumented client. You can organize and group subsegments by wrapping client calls in custom subsegments. You can create a custom subsegment for an entire function or any section of code, and record metadata and annotations on the subsegment instead of writing everything on the parent segment.

For reference documentation about the SDK's classes and methods, see the AWS X-Ray SDK for Node.js API Reference.

Requirements

The X-Ray SDK for Node.js requires Node.js and the following libraries:

- `atomic-batcher` – 1.0.2
- `cls-hooked` – 4.2.2
- `pkginfo` – 0.4.0
- `semver` – 5.3.0

The SDK pulls these libraries in when you install it with NPM.

To trace AWS SDK clients, the X-Ray SDK for Node.js requires a minimum version of the AWS SDK for JavaScript in Node.js.

- `aws-sdk` – 2.7.15

Dependency management

The X-Ray SDK for Node.js is available from NPM.

- **Package** – `aws-xray-sdk`

For local development, install the SDK in your project directory with npm.

```
~/nodejs-xray$ npm install aws-xray-sdk
aws-xray-sdk@3.3.3
  ### aws-xray-sdk-core@3.3.3
  # ### @aws-sdk/service-error-classification@3.15.0
  # ### @aws-sdk/types@3.15.0
  # ### @types/cls-hooked@4.3.3
  # # ### @types/node@15.3.0
  # ### atomic-batcher@1.0.2
  # ### cls-hooked@4.2.2
  # # ### async-hook-jl@1.7.6
  # # # ### stack-chain@1.3.7
  # # ### emitter-listener@1.1.2
  # #   ### shimmer@1.2.1
  # ### semver@5.7.1
  ### aws-xray-sdk-express@3.3.3
  ### aws-xray-sdk-mysql@3.3.3
  ### aws-xray-sdk-postgres@3.3.3
```

Use the `--save` option to save the SDK as a dependency in your application's `package.json`.

```
~/nodejs-xray$ npm install aws-xray-sdk --save
aws-xray-sdk@3.3.3
```

If your application has any dependencies whose versions conflict with the X-Ray SDK's dependencies, both versions will be installed to ensure compatibility. For more details, see the official NPM documentation for dependency resolution.

Node.js samples

Work with the AWS X-Ray SDK for Node.js to get an end-to-end view of requests as they travel through your Node.js applications.

- Node.js sample application on GitHub.

Configuring the X-Ray SDK for Node.js

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can configure the X-Ray SDK for Node.js with plugins to include information about the service that your application runs on, modify the default sampling behavior, or add sampling rules that apply to requests to specific paths.

Sections

- Service plugins
- Sampling rules
- Logging
- X-Ray daemon address
- Environment variables

Service plugins

Use `plugins` to record information about the service hosting your application.

Plugins

- Amazon EC2 – `EC2Plugin` adds the instance ID, Availability Zone, and the CloudWatch Logs Group.
- Elastic Beanstalk – `ElasticBeanstalkPlugin` adds the environment name, version label, and deployment ID.
- Amazon ECS – `ECSPlugin` adds the container ID.

To use a plugin, configure the X-Ray SDK for Node.js client by using the `config` method.

Example app.js - plugins

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.config([AWSXRay.plugins.EC2Plugin,AWSXRay.plugins.ElasticBeanstalkPlugin]);
```

The SDK also uses plugin settings to set the `origin` field on the segment. This indicates the type of AWS resource that runs your application. When you use multiple plugins, the SDK uses the following resolution order to determine the origin: ElasticBeanstalk > EKS > ECS > EC2.

Sampling rules

The SDK uses the sampling rules you define in the X-Ray console to determine which requests to record. The default rule traces the first request each second, and five percent of any additional requests across all services sending traces to X-Ray. Create additional rules in the X-Ray console to customize the amount of data recorded for each of your applications.

The SDK applies custom rules in the order in which they are defined. If a request matches multiple custom rules, the SDK applies only the first rule.

> **Note**
>
> If the SDK can't reach X-Ray to get sampling rules, it reverts to a default local rule of the first request each second, and five percent of any additional requests per host. This can occur if the host doesn't have permission to call sampling APIs, or can't connect to the X-Ray daemon, which acts as a TCP proxy for API calls made by the SDK.

You can also configure the SDK to load sampling rules from a JSON document. The SDK can use local rules as a backup for cases where X-Ray sampling is unavailable, or use local rules exclusively.

Example sampling-rules.json

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

This example defines one custom rule and a default rule. The custom rule applies a five-percent sampling rate with no minimum number of requests to trace for paths under `/api/move/`. The default rule traces the first request each second and 10 percent of additional requests.

The disadvantage of defining rules locally is that the fixed target is applied by each instance of the recorder independently, instead of being managed by the X-Ray service. As you deploy more hosts, the fixed rate is multiplied, making it harder to control the amount of data recorded.

On AWS Lambda, you cannot modify the sampling rate. If your function is called by an instrumented service, calls that generated requests that were sampled by that service will be recorded by Lambda. If active tracing is enabled and no tracing header is present, Lambda makes the sampling decision.

To configure backup rules, tell the X-Ray SDK for Node.js to load sampling rules from a file with `setSamplingRules`.

Example app.js - sampling rules from a file

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.middleware.setSamplingRules('sampling-rules.json');
```

You can also define your rules in code and pass them to `setSamplingRules` as an object.

Example app.js - sampling rules from an object

```
var AWSXRay = require('aws-xray-sdk');
var rules = {
  "rules": [ { "description": "Player moves.", "service_name": "*", "http_method": "*",
 "url_path": "/api/move/*", "fixed_target": 0, "rate": 0.05 } ],
  "default": { "fixed_target": 1, "rate": 0.1 },
  "version": 1
  }

AWSXRay.middleware.setSamplingRules(rules);
```

To use only local rules, call `disableCentralizedSampling`.

```
AWSXRay.middleware.disableCentralizedSampling()
```

Logging

To log output from the SDK, call `AWSXRay.setLogger(logger)`, where `logger` is an object that provides standard logging methods (`warn`, `info`, etc.).

By default the SDK will log error messages to the console using the standard methods on the console object. The log level of the built-in logger can be set by using either the `AWS_XRAY_DEBUG_MODE` or `AWS_XRAY_LOG_LEVEL` environment variables. For a list of valid log level values, see Environment variables.

If you wish to provide a different format or destination for the logs then you can provide the SDK with your own implementation of the logger interface as shown below. Any object that implements this interface can be used. This means that many logging libraries, e.g. Winston, could be used and passed to the SDK directly.

Example app.js - logging

```
var AWSXRay = require('aws-xray-sdk');

// Create your own logger, or instantiate one using a library.
var logger = {
  error: (message, meta) => { /* logging code */ },
  warn: (message, meta) => { /* logging code */ },
  info: (message, meta) => { /* logging code */ },
  debug: (message, meta) => { /* logging code */ }
}

AWSXRay.setLogger(logger);
AWSXRay.config([AWSXRay.plugins.EC2Plugin]);
```

Call `setLogger` before you run other configuration methods to ensure that you capture output from those operations.

X-Ray daemon address

If the X-Ray daemon listens on a port or host other than `127.0.0.1:2000`, you can configure the X-Ray SDK for Node.js to send trace data to a different address.

```
AWSXRay.setDaemonAddress('host:port');
```

You can specify the host by name or by IPv4 address.

Example app.js - daemon address

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.setDaemonAddress('daemonhost:8082');
```

If you configured the daemon to listen on different ports for TCP and UDP, you can specify both in the daemon address setting.

Example app.js - daemon address on separate ports

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.setDaemonAddress('tcp:daemonhost:8082 udp:daemonhost:8083');
```

You can also set the daemon address by using the `AWS_XRAY_DAEMON_ADDRESS` environment variable.

Environment variables

You can use environment variables to configure the X-Ray SDK for Node.js. The SDK supports the following variables.

- `AWS_XRAY_CONTEXT_MISSING` – Set to `RUNTIME_ERROR` to throw exceptions when your instrumented code attempts to record data when no segment is open.

  Valid Values

  - `RUNTIME_ERROR` – Throw a runtime exception.
  - `LOG_ERROR` – Log an error and continue (default).
  - `IGNORE_ERROR` – Ignore error and continue.

Errors related to missing segments or subsegments can occur when you attempt to use an instrumented client in startup code that runs when no request is open, or in code that spawns a new thread.

- `AWS_XRAY_DAEMON_ADDRESS` – Set the host and port of the X-Ray daemon listener. By default, the SDK uses `127.0.0.1:2000` for both trace data (UDP) and sampling (TCP). Use this variable if you have configured the daemon to listen on a different port or if it is running on a different host.

  Format

  - **Same port** – `address:port`
  - **Different ports** – `tcp:address:port udp:address:port`
- `AWS_XRAY_DEBUG_MODE` – Set to `TRUE` to configure the SDK to output logs to the console, at `debug` level.
- `AWS_XRAY_LOG_LEVEL` – Set a log level for the default logger. Valid values are `debug`, `info`, `warn`, `error`, and `silent`. This value is ignored when AWS_XRAY_DEBUG_MODE is set to `TRUE`.
- `AWS_XRAY_TRACING_NAME` – Set a service name that the SDK uses for segments. Overrides the segment name that you set on the Express middleware.

Tracing incoming requests with the X-Ray SDK for Node.js

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can use the X-Ray SDK for Node.js to trace incoming HTTP requests that your Express and Restify applications serve on an EC2 instance in Amazon EC2, AWS Elastic Beanstalk, or Amazon ECS.

The X-Ray SDK for Node.js provides middleware for applications that use the Express and Restify frameworks. When you add the X-Ray middleware to your application, the X-Ray SDK for Node.js creates a segment for each sampled request. This segment includes timing, method, and disposition of the HTTP request. Additional instrumentation creates subsegments on this segment.

> **Note**
>
> For AWS Lambda functions, Lambda creates a segment for each sampled request. See AWS Lambda and AWS X-Ray for more information.

Each segment has a name that identifies your application in the service map. The segment can be named statically, or you can configure the SDK to name it dynamically based on the host header in the incoming request. Dynamic naming lets you group traces based on the domain name in the request, and apply a default name if the name doesn't match an expected pattern (for example, if the host header is forged).

> **Forwarded Requests**
>
> If a load balancer or other intermediary forwards a request to your application, X-Ray takes the client IP from the `X-Forwarded-For` header in the request instead of from the source IP in the IP packet. The client IP that is recorded for a forwarded request can be forged, so it should not be trusted.

When a request is forwarded, the SDK sets an additional field in the segment to indicate this. If the segment contains the field `x_forwarded_for` set to `true`, the client IP was taken from the `X-Forwarded-For` header in the HTTP request.

The message handler creates a segment for each incoming request with an `http` block that contains the following information:

- **HTTP method** – GET, POST, PUT, DELETE, etc.
- **Client address** – The IP address of the client that sent the request.
- **Response code** – The HTTP response code for the completed request.
- **Timing** – The start time (when the request was received) and end time (when the response was sent).

- **User agent** — The `user-agent` from the request.
- **Content length** — The `content-length` from the response.

Sections

- Tracing incoming requests with Express
- Tracing incoming requests with restify
- Configuring a segment naming strategy

Tracing incoming requests with Express

To use the Express middleware, initialize the SDK client and use the middleware returned by the `express.openSegment` function before you define your routes.

Example app.js - Express

```
var app = express();

var AWSXRay = require('aws-xray-sdk');
app.use(AWSXRay.express.openSegment('MyApp'));

app.get('/', function (req, res) {
  res.render('index');
});

app.use(AWSXRay.express.closeSegment());
```

After you define your routes, use the output of `express.closeSegment` as shown to handle any errors returned by the X-Ray SDK for Node.js.

Tracing incoming requests with restify

To use the Restify middleware, initialize the SDK client and run `enable`. Pass it your Restify server and segment name.

Example app.js - restify

```
var AWSXRay = require('aws-xray-sdk');
var AWSXRayRestify = require('aws-xray-sdk-restify');
var restify = require('restify');
var server = restify.createServer();
AWSXRayRestify.enable(server, 'MyApp'));

server.get('/', function (req, res) {
  res.render('index');
});
```

Configuring a segment naming strategy

AWS X-Ray uses a *service name* to identify your application and distinguish it from the other applications, databases, external APIs, and AWS resources that your application uses. When the X-Ray SDK generates segments for incoming requests, it records your application's service name in the segment's name field.

The X-Ray SDK can name segments after the hostname in the HTTP request header. However, this header can be forged, which could result in unexpected nodes in your service map. To prevent the SDK from naming segments incorrectly due to requests with forged host headers, you must specify a default name for incoming requests.

If your application serves requests for multiple domains, you can configure the SDK to use a dynamic naming strategy to reflect this in segment names. A dynamic naming strategy allows the SDK to use the hostname for requests that match an expected pattern, and apply the default name to requests that don't.

For example, you might have a single application serving requests to three subdomains– `www.example.com`, `api.example.com`, and `static.example.com`. You can use a dynamic naming strategy with the pattern `*.example.com` to identify segments for each subdomain with a different name, resulting in three service nodes on the service map. If your application receives requests with a hostname that doesn't match the pattern, you will see a fourth node on the service map with a fallback name that you specify.

To use the same name for all request segments, specify the name of your application when you initialize the middleware, as shown in the previous sections.

Note

You can override the default service name that you define in code with the `AWS_XRAY_TRACING_NAME` environment variable.

A dynamic naming strategy defines a pattern that hostnames should match, and a default name to use if the hostname in the HTTP request does not match the pattern. To name segments dynamically, use `AWSXRay.middleware.enableDynamicNaming`.

Example app.js - dynamic segment names

If the hostname in the request matches the pattern `*.example.com`, use the hostname. Otherwise, use `MyApp`.

```
var app = express();

var AWSXRay = require('aws-xray-sdk');
app.use(AWSXRay.express.openSegment('MyApp'));
AWSXRay.middleware.enableDynamicNaming('*.example.com');

app.get('/', function (req, res) {
  res.render('index');
});

app.use(AWSXRay.express.closeSegment());
```

Tracing AWS SDK calls with the X-Ray SDK for Node.js

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

When your application makes calls to AWS services to store data, write to a queue, or send notifications, the X-Ray SDK for Node.js tracks the calls downstream in subsegments. Traced AWS services, and resources that you access within those services (for example, an Amazon S3 bucket or Amazon SQS queue), appear as downstream nodes on the trace map in the X-Ray console.

Instrument AWS SDK clients that you create via the AWS SDK for JavaScript V2 or AWS SDK for JavaScript V3. Each AWS SDK version provides different methods for instrumenting AWS SDK clients.

Note

Currently, the AWS X-Ray SDK for Node.js returns less segment information when instrumenting AWS SDK for JavaScript V3 clients, as compared to instrumenting V2 clients. For instance, subsegments representing calls to DynamoDB will not return the table name. If you need this segment information in your traces, consider using the AWS SDK for JavaScript V2.

AWS SDK for JavaScript V2

You can instrument all AWS SDK V2 clients by wrapping your `aws-sdk` require statement in a call to `AWSXRay.captureAWS`.

Example app.js - AWS SDK instrumentation

```
const AWS = AWSXRay.captureAWS(require('aws-sdk'));
```

To instrument individual clients, wrap your AWS SDK client in a call to `AWSXRay.captureAWSClient`. For example, to instrument an `AmazonDynamoDB` client:

Example app.js - DynamoDB client instrumentation

```
    const AWSXRay = require('aws-xray-sdk');
...
    const ddb = AWSXRay.captureAWSClient(new AWS.DynamoDB());
```

Warning

Do not use both `captureAWS` and `captureAWSClient` together. This will lead to duplicate subsegments.

If you want to use TypeScript with ECMAScript modules (ESM) to load your JavaScript code, use the following example to import libraries:

Example app.js - AWS SDK instrumentation

```
import * as AWS from 'aws-sdk';
import * as AWSXRay from 'aws-xray-sdk';
```

To instrument all AWS clients with ESM, use the following code:

Example app.js - AWS SDK instrumentation

```
import * as AWS from 'aws-sdk';
import * as AWSXRay from 'aws-xray-sdk';
const XRAY_AWS = AWSXRay.captureAWS(AWS);
const ddb = new XRAY_AWS.DynamoDB();
```

For all services, you can see the name of the API called in the X-Ray console. For a subset of services, the X-Ray SDK adds information to the segment to provide more granularity in the service map.

For example, when you make a call with an instrumented DynamoDB client, the SDK adds the table name to the segment for calls that target a table. In the console, each table appears as a separate node in the service map, with a generic DynamoDB node for calls that don't target a table.

Example Subsegment for a call to DynamoDB to save an item

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
 }
}
```

When you access named resources, calls to the following services create additional nodes in the service map. Calls that don't target specific resources create a generic node for the service.

- **Amazon DynamoDB** – Table name
- **Amazon Simple Storage Service** – Bucket and key name
- **Amazon Simple Queue Service** – Queue name

AWS SDK for JavaScript V3

The AWS SDK for JavaScript V3 is modular, so your code only loads the modules it needs. Because of this, it isn't possible to instrument all AWS SDK clients as V3 does not support the `captureAWS` method.

If you want to use TypeScript with ECMAScript Modules (ESM) to load your JavaScript code, you can use the following example to import libraries:

```
import * as AWS from 'aws-sdk';
import * as AWSXRay from 'aws-xray-sdk';
```

Instrument each AWS SDK client using the `AWSXRay.captureAWSv3Client` method. For example, to instrument an `AmazonDynamoDB` client:

Example app.js - DynamoDB client instrumentation using SDK for Javascript V3

```
  const AWSXRay = require('aws-xray-sdk');
  const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
...
  const ddb = AWSXRay.captureAWSv3Client(new DynamoDBClient({ region:
 "region" }));
```

When using AWS SDK for JavaScript V3, metadata such as table name, bucket and key name, or queue name, are not currently returned, and therefore the trace map will not contain discrete nodes for each named resource as it would when instrumenting AWS SDK clients using the AWS SDK for JavaScript V2.

Example Subsegment for a call to DynamoDB to save an item, when using the AWS SDK for JavaScript V3

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

Tracing calls to downstream HTTP web services using the X-Ray SDK for Node.js

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

When your application makes calls to microservices or public HTTP APIs, you can use the X-Ray SDK for Node.js client to instrument those calls and add the API to the service graph as a downstream service.

Pass your `http` or `https` client to the X-Ray SDK for Node.js `captureHTTPs` method to trace outgoing calls.

> **Note**
>
> Calls using third-party HTTP request libraries, such as Axios or Superagent, are supported through the [`captureHTTPsGlobal()` API]() and will still be traced when they use the native `http` module.

Example app.js - HTTP client

```
var AWSXRay = require('aws-xray-sdk');
var http = AWSXRay.captureHTTPs(require('http'));
```

To enable tracing on all HTTP clients, call `captureHTTPsGlobal` before you load `http`.

Example app.js - HTTP client (global)

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.captureHTTPsGlobal(require('http'));
var http = require('http');
```

When you instrument a call to a downstream web API, the X-Ray SDK for Node.js records a subsegment that contains information about the HTTP request and response. X-Ray uses the subsegment to generate an inferred segment for the remote API.

Example Subsegment for a downstream HTTP call

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

Example Inferred segment for a downstream HTTP call

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
  "trace_id": "1-62be1272-1b71c4274f39f122afa64eab",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  },
  "inferred": true
}
```

Tracing SQL queries with the X-Ray SDK for Node.js

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

Instrument SQL database queries by wrapping your SQL client in the corresponding X-Ray SDK for Node.js client method.

- **PostgreSQL** – `AWSXRay.capturePostgres()`

  ```
  var AWSXRay = require('aws-xray-sdk');
  var pg = AWSXRay.capturePostgres(require('pg'));
  var client = new pg.Client();
  ```

- **MySQL** – `AWSXRay.captureMySQL()`

  ```
  var AWSXRay = require('aws-xray-sdk');
  var mysql = AWSXRay.captureMySQL(require('mysql'));
  ...
  var connection = mysql.createConnection(config);
  ```

When you use an instrumented client to make SQL queries, the X-Ray SDK for Node.js records information about the connection and query in a subsegment.

Including additional data in SQL subsegments

You can add additional information to subsegments generated for SQL queries, as long as it's mapped to an allow-listed SQL field. For example, to record the sanitized SQL query string in a subsegment, you can add it directly to the subsegment's SQL object.

Example Assign SQL to subsegment

```
    const queryString = 'SELECT * FROM MyTable';
connection.query(queryString, ...);

// Retrieve the most recently created subsegment
const subs = AWSXRay.getSegment().subsegments;

if (subs & & subs.length > 0) {
  var sqlSub = subs[subs.length - 1];
  sqlSub.sql.sanitized_query = queryString;
}
```

For a full list of allow-listed SQL fields, see SQL Queries in the *AWS X-Ray Developer Guide*.

Generating custom subsegments with the X-Ray SDK for Node.js

Subsegments extend a trace's segment with details about work done in order to serve a request. Each time you make a call with an instrumented client, the X-Ray SDK records the information generated in a subsegment. You can create additional subsegments to group other subsegments, to measure the performance of a section of code, or to record annotations and metadata.

Custom Express subsegments

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

To create a custom subsegment for a function that makes calls to downstream services, use the `captureAsyncFunc` function.

Example app.js - custom subsegments Express

```
var AWSXRay = require('aws-xray-sdk');

app.use(AWSXRay.express.openSegment('MyApp'));

app.get('/', function (req, res) {
  var host = 'api.example.com';

  AWSXRay.captureAsyncFunc('send', function(subsegment) {
    sendRequest(host, function() {
      console.log('rendering!');
      res.render('index');
      subsegment.close();
    });
  });
});

app.use(AWSXRay.express.closeSegment());
```

```
function sendRequest(host, cb) {
  var options = {
    host: host,
    path: '/',
  };

  var callback = function(response) {
    var str = '';

    response.on('data', function (chunk) {
      str += chunk;
    });

    response.on('end', function () {
      cb();
    });
  }

  http.request(options, callback).end();
};
```

In this example, the application creates a custom subsegment named `send` for calls to the `sendRequest` function. `captureAsyncFunc` passes a subsegment that you must close within the callback function when the asynchronous calls that it makes are complete.

For synchronous functions, you can use the `captureFunc` function, which closes the subsegment automatically as soon as the function block finishes executing.

When you create a subsegment within a segment or another subsegment, the X-Ray SDK for Node.js generates an ID for it and records the start time and end time.

Example Subsegment with metadata

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
```

Custom Lambda subsegments

The SDK is configured to automatically create a placeholder facade segment when it detects it's running in Lambda. To create a basic subsegement, which will create a single `AWS::Lambda::Function` node on the X-Ray trace map, call and repurpose the facade segment. If you manually create a new segment with a new ID (while sharing the trace ID, parent ID and the sampling decision) you will be able to send a new segment.

Example app.js - manual custom subsegments

```
const segment = AWSXRay.getSegment(); //returns the facade segment
const subsegment = segment.addNewSubsegment('subseg');
...
subsegment.close();
//the segment is closed by the SDK automatically
```

Add annotations and metadata to segments with the X-Ray SDK for Node.js

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can record additional information about requests, the environment, or your application with annotations and metadata. You can add annotations and metadata to the segments that the X-Ray SDK creates, or to custom subsegments that you create.

Annotations are key-value pairs with string, number, or Boolean values. Annotations are indexed for use with filter expressions. Use annotations to record data that you want to use to group traces in the console, or when calling the `GetTraceSummaries` API.

Metadata are key-value pairs that can have values of any type, including objects and lists, but are not indexed for use with filter expressions. Use metadata to record additional data that you want stored in the trace but don't need to use with search.

In addition to annotations and metadata, you can also record user ID strings on segments. User IDs are recorded in a separate field on segments and are indexed for use with search.

Sections

- Recording annotations with the X-Ray SDK for Node.js
- Recording metadata with the X-Ray SDK for Node.js
- Recording user IDs with the X-Ray SDK for Node.js

Recording annotations with the X-Ray SDK for Node.js

Use annotations to record information on segments or subsegments that you want indexed for search.

Annotation Requirements

- **Keys** – The key for an X-Ray annotation can have up to 500 alphanumeric characters. You cannot use spaces or symbols other than a dot or period (.)
- **Values** – The value for an X-Ray annotation can have up to 1,000 Unicode characters.
- The number of **Annotations** – You can use up to 50 annotations per trace.

To record annotations

1. Get a reference to the current segment or subsegment.

```
var AWSXRay = require('aws-xray-sdk');
...
var document = AWSXRay.getSegment();
```

2. Call `addAnnotation` with a String key, and a Boolean, Number, or String value.

```
document.addAnnotation("mykey", "my value");
```

The following example shows how to call `putAnnotation` with a String key that includes a dot, and a Boolean, Number, or String value.

```
document.putAnnotation("testkey.test", "my value");
```

The SDK records annotations as key-value pairs in an `annotations` object in the segment document. Calling `addAnnotation` twice with the same key overwrites previously recorded values on the same segment or subsegment.

To find traces that have annotations with specific values, use the `annotation[key]` keyword in a filter expression.

Example app.js - annotations

```
var AWS = require('aws-sdk');
var AWSXRay = require('aws-xray-sdk');
var ddb = AWSXRay.captureAWSClient(new AWS.DynamoDB());
...
app.post('/signup', function(req, res) {
    var item = {
        'email': {'S': req.body.email},
        'name': {'S': req.body.name},
        'preview': {'S': req.body.previewAccess},
        'theme': {'S': req.body.theme}
    };

    var seg = AWSXRay.getSegment();
    seg.addAnnotation('theme', req.body.theme);

    ddb.putItem({
      'TableName': ddbTable,
      'Item': item,
      'Expected': { email: { Exists: false } }
  }, function(err, data) {
...
```

Recording metadata with the X-Ray SDK for Node.js

Use metadata to record information on segments or subsegments that you don't need indexed for search. Metadata values can be strings, numbers, Booleans, or any other object that can be serialized into a JSON object or array.

To record metadata

1. Get a reference to the current segment or subsegment.

```
var AWSXRay = require('aws-xray-sdk');
...
var document = AWSXRay.getSegment();
```

2. Call `addMetadata` with a string key, a Boolean, number, string, or object value, and a string namespace.

```
document.addMetadata("my key", "my value", "my namespace");
```

   or

   Call `addMetadata` with just a key and value.

```
document.addMetadata("my key", "my value");
```

If you don't specify a namespace, the SDK uses `default`. Calling `addMetadata` twice with the same key overwrites previously recorded values on the same segment or subsegment.

Recording user IDs with the X-Ray SDK for Node.js

Record user IDs on request segments to identify the user who sent the request. This operation isn’t compatible with AWS Lambda functions because segments in Lambda environments are immutable. The `setUser` call can be applied only to segments, not subsegments.

To record user IDs

1. Get a reference to the current segment or subsegment.

```
var AWSXRay = require('aws-xray-sdk');
...
var document = AWSXRay.getSegment();
```

2. Call `setUser()` with a string ID of the user who sent the request.

```
var user = 'john123';
```

```
AWSXRay.getSegment().setUser(user);
```

You can call `setUser` to record the user ID as soon as your express application starts processing a request. If you will use the segment only to set the user ID, you can chain the calls in a single line.

Example app.js - user ID

```
var AWS = require('aws-sdk');
var AWSXRay = require('aws-xray-sdk');
var uuidv4 = require('uuid/v4');
var ddb = AWSXRay.captureAWSClient(new AWS.DynamoDB());
...
    app.post('/signup', function(req, res) {
    var userId = uuidv4();
    var item = {
        'userId': {'S': userId},
        'email': {'S': req.body.email},
        'name': {'S': req.body.name}
    };

    var seg = AWSXRay.getSegment().setUser(userId);

    ddb.putItem({
      'TableName': ddbTable,
      'Item': item,
      'Expected': { email: { Exists: false } }
  }, function(err, data) {
...
```

To find traces for a user ID, use the `user` keyword in a filter expression.

Working with Python

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

There are two ways to instrument your Python application to send traces to X-Ray:

- AWS Distro for OpenTelemetry Python – An AWS distribution that provides a set of open source libraries for sending correlated metrics and traces to multiple AWS monitoring solutions, including Amazon CloudWatch, AWS X-Ray, and Amazon OpenSearch Service, via the AWS Distro for OpenTelemetry Collector.
- AWS X-Ray SDK for Python – A set of libraries for generating and sending traces to X-Ray via the X-Ray daemon.

For more information, see Choosing between the AWS Distro for OpenTelemetry and X-Ray SDKs.

AWS Distro for OpenTelemetry Python

With the AWS Distro for OpenTelemetry (ADOT) Python, you can instrument your applications once and send correlated metrics and traces to multiple AWS monitoring solutions including Amazon CloudWatch, AWS X-Ray, and Amazon OpenSearch Service. Using X-Ray with ADOT requires two components: an *OpenTelemetry SDK* enabled for use with X-Ray, and the *AWS Distro for OpenTelemetry Collector* enabled for use with X-Ray. ADOT Python includes auto-instrumentation support, enabling your application to send traces without code changes.

To get started, see the AWS Distro for OpenTelemetry Python documentation.

For more information about using the AWS Distro for OpenTelemetry with AWS X-Ray and other AWS services, see AWS Distro for OpenTelemetry or the AWS Distro for OpenTelemetry Documentation.

For more information about language support and usage, see AWS Observability on GitHub.

AWS X-Ray SDK for Python

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The X-Ray SDK for Python is a library for Python web applications that provides classes and methods for generating and sending trace data to the X-Ray daemon. Trace data includes information about incoming HTTP requests served by the application, and calls that the application makes to downstream services using the AWS SDK, HTTP clients, or an SQL database connector. You can also create segments manually and add debug information in annotations and metadata.

You can download the SDK with `pip`.

```
$ pip install aws-xray-sdk
```

Note

The X-Ray SDK for Python is an open source project. You can follow the project and submit issues and pull requests on GitHub: github.com/aws/aws-xray-sdk-python

If you use Django or Flask, start by adding the SDK middleware to your application to trace incoming requests. The middleware creates a segment for each traced request, and completes the segment when the response is sent. While the segment is open, you can use the SDK client's methods to add information to the segment and create subsegments to trace downstream calls. The SDK also automatically records exceptions that your application throws while the segment is open. For other applications, you can create segments manually.

For Lambda functions called by an instrumented application or service, Lambda reads the tracing header and traces sampled requests automatically. For other functions, you can configure Lambda to sample and trace incoming requests. In either case, Lambda creates the segment and provides it to the X-Ray SDK.

> **Note**
>
> On Lambda, the X-Ray SDK is optional. If you don't use it in your function, your service map will still include a node for the Lambda service, and one for each Lambda function. By adding the SDK, you can instrument your function code to add subsegments to the function segment recorded by Lambda. See AWS Lambda and AWS X-Ray for more information.

See Worker for a example Python function instrumented in Lambda.

Next, use the X-Ray SDK for Python to instrument downstream calls by patching your application's libraries. The SDK supports the following libraries.

Supported Libraries

- `botocore`, `boto3` – Instrument AWS SDK for Python (Boto) clients.
- `pynamodb` – Instrument PynamoDB's version of the Amazon DynamoDB client.
- `aiobotocore`, `aioboto3` – Instrument asyncio-integrated versions of SDK for Python clients.
- `requests`, `aiohttp` – Instrument high-level HTTP clients.
- `httplib`, `http.client` – Instrument low-level HTTP clients and the higher level libraries that use them.
- `sqlite3` – Instrument SQLite clients.
- `mysql-connector-python` – Instrument MySQL clients.
- `pg8000` – Instrument Pure-Python PostgreSQL interface.
- `psycopg2` – Instrument PostgreSQL database adapter.
- `pymongo` – Instrument MongoDB clients.
- `pymysql` – Instrument PyMySQL based clients for MySQL and MariaDB.

Whenever your application makes calls to AWS, an SQL database, or other HTTP services, the SDK records information about the call in a subsegment. AWS services and the resources that you access within the services appear as downstream nodes on the trace map to help you identify errors and throttling issues on individual connections.

After you start using the SDK, customize its behavior by configuring the recorder and middleware. You can add plugins to record data about the compute resources running your application, customize sampling behavior by defining sampling rules, and set the log level to see more or less information from the SDK in your application logs.

Record additional information about requests and the work that your application does in annotations and metadata. Annotations are simple key-value pairs that are indexed for use with filter expressions, so that you can search for traces that contain specific data. Metadata entries are less restrictive and can record entire objects and arrays — anything that can be serialized into JSON.

> **Annotations and Metadata**
>
> Annotations and metadata are arbitrary text that you add to segments with the X-Ray SDK. Annotations are indexed for use with filter expressions. Metadata are not indexed, but can be viewed in the raw segment with the X-Ray console or API. Anyone that you grant read access to X-Ray can view this data.

When you have a lot of instrumented clients in your code, a single request segment can contain a large number of subsegments, one for each call made with an instrumented client. You can organize and group subsegments by wrapping client calls in custom subsegments. You can create a custom subsegment for an entire function or any section of code. You can then you can record metadata and annotations on the subsegment instead of writing everything on the parent segment.

For reference documentation for the SDK's classes and methods, see the AWS X-Ray SDK for Python API Reference.

Requirements

The X-Ray SDK for Python supports the following language and library versions.

- **Python** – 2.7, 3.4, and newer
- **Django** – 1.10 and newer

- **Flask** – 0.10 and newer
- **aiohttp** – 2.3.0 and newer
- **AWS SDK for Python (Boto)** – 1.4.0 and newer
- **botocore** – 1.5.0 and newer
- **enum** – 0.4.7 and newer, for Python versions 3.4.0 and older
- **jsonpickle** – 1.0.0 and newer
- **setuptools** – 40.6.3 and newer
- **wrapt** – 1.11.0 and newer

Dependency management

The X-Ray SDK for Python is available from `pip`.

- **Package** – `aws-xray-sdk`

Add the SDK as a dependency in your `requirements.txt` file.

Example requirements.txt

```
aws-xray-sdk==2.4.2
boto3==1.4.4
botocore==1.5.55
Django==1.11.3
```

If you use Elastic Beanstalk to deploy your application, Elastic Beanstalk installs all of the packages in `requirements.txt` automatically.

Configuring the X-Ray SDK for Python

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more

information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The X-Ray SDK for Python has a class named xray_recorder that provides the global recorder. You can configure the global recorder to customize the middleware that creates segments for incoming HTTP calls.

Sections

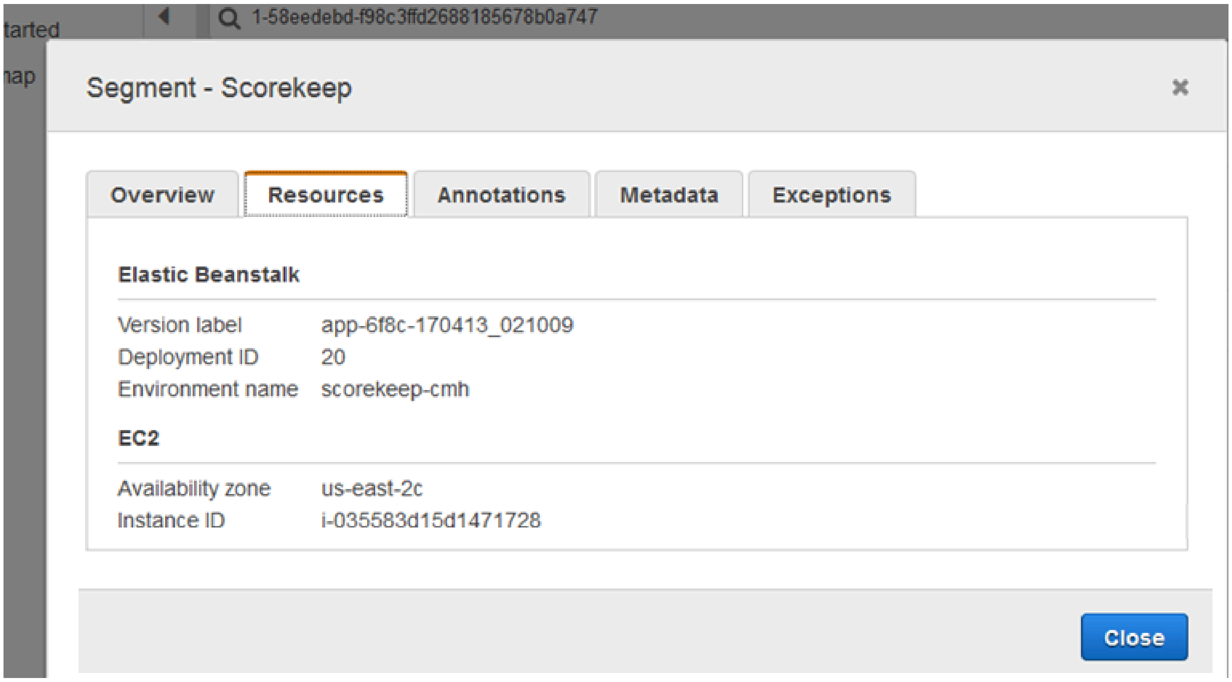
- Service plugins
- Sampling rules
- Logging
- Recorder configuration in code
- Recorder configuration with Django
- Environment variables

Service plugins

Use plugins to record information about the service hosting your application.

Plugins

- Amazon EC2 – EC2Plugin adds the instance ID, Availability Zone, and the CloudWatch Logs Group.
- Elastic Beanstalk – ElasticBeanstalkPlugin adds the environment name, version label, and deployment ID.
- Amazon ECS – ECSPlugin adds the container ID.



Segment - Scorekeep

Elastic Beanstalk

Version label	app-6f8c-170413_021009
Deployment ID	20
Environment name	scorekeep-cmh

EC2

Availability zone	us-east-2c
Instance ID	i-035583d15d1471728

To use a plugin, call `configure` on the `xray_recorder`.

```
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

xray_recorder.configure(service='My app')
plugins = ('ElasticBeanstalkPlugin', 'EC2Plugin')
xray_recorder.configure(plugins=plugins)
patch_all()
```

Note

Since `plugins` are passed in as a tuple, be sure to include a trailing `,` when specifying a single plugin. For example, `plugins = ('EC2Plugin',)`

You can also use environment variables, which take precedence over values set in code, to configure the recorder.

Configure plugins before patching libraries to record downstream calls.

The SDK also uses plugin settings to set the `origin` field on the segment. This indicates the type of AWS resource that runs your application. When you use multiple plugins, the SDK uses the following resolution order to determine the origin: ElasticBeanstalk > EKS > ECS > EC2.

Sampling rules

The SDK uses the sampling rules you define in the X-Ray console to determine which requests to record. The default rule traces the first request each second, and five percent of any additional requests across all services sending traces to X-Ray. Create additional rules in the X-Ray console to customize the amount of data recorded for each of your applications.

The SDK applies custom rules in the order in which they are defined. If a request matches multiple custom rules, the SDK applies only the first rule.

> **Note**
>
> If the SDK can't reach X-Ray to get sampling rules, it reverts to a default local rule of the first request each second, and five percent of any additional requests per host. This can occur if the host doesn't have permission to call sampling APIs, or can't connect to the X-Ray daemon, which acts as a TCP proxy for API calls made by the SDK.

You can also configure the SDK to load sampling rules from a JSON document. The SDK can use local rules as a backup for cases where X-Ray sampling is unavailable, or use local rules exclusively.

Example sampling-rules.json

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

This example defines one custom rule and a default rule. The custom rule applies a five-percent sampling rate with no minimum number of requests to trace for paths under `/api/move/`. The default rule traces the first request each second and 10 percent of additional requests.

The disadvantage of defining rules locally is that the fixed target is applied by each instance of the recorder independently, instead of being managed by the X-Ray service. As you deploy more hosts, the fixed rate is multiplied, making it harder to control the amount of data recorded.

On AWS Lambda, you cannot modify the sampling rate. If your function is called by an instrumented service, calls that generated requests that were sampled by that service will be recorded by Lambda. If active tracing is enabled and no tracing header is present, Lambda makes the sampling decision.

To configure backup sampling rules, call `xray_recorder.configure`, as shown in the following example, where *`rules`* is either a dictionary of rules or the absolute path to a JSON file containing sampling rules.

```
xray_recorder.configure(sampling_rules=rules)
```

To use only local rules, configure the recorder with a `LocalSampler`.

```
from aws_xray_sdk.core.sampling.local.sampler import LocalSampler
xray_recorder.configure(sampler=LocalSampler())
```

You can also configure the global recorder to disable sampling and instrument all incoming requests.

Example main.py – Disable sampling

```
xray_recorder.configure(sampling=False)
```

Logging

The SDK uses Python's built-in `logging` module with a default `WARNING` logging level. Get a reference to the logger for the `aws_xray_sdk` class and call `setLevel` on it to configure the different log level for the library and the rest of your application.

Example app.py – Logging

```
logging.basicConfig(level='WARNING')
logging.getLogger('aws_xray_sdk').setLevel(logging.ERROR)
```

Use debug logs to identify issues, such as unclosed subsegments, when you generate subsegments manually.

Recorder configuration in code

Additional settings are available from the `configure` method on `xray_recorder`.

- `context_missing` – Set to `LOG_ERROR` to avoid throwing exceptions when your instrumented code attempts to record data when no segment is open.
- `daemon_address` – Set the host and port of the X-Ray daemon listener.
- `service` – Set a service name that the SDK uses for segments.
- `plugins` – Record information about your application's AWS resources.
- `sampling` – Set to `False` to disable sampling.
- `sampling_rules` – Set the path of the JSON file containing your sampling rules.

Example main.py – Disable context missing exceptions

```
from aws_xray_sdk.core import xray_recorder

xray_recorder.configure(context_missing='LOG_ERROR')
```

Recorder configuration with Django

If you use the Django framework, you can use the Django `settings.py` file to configure options on the global recorder.

- `AUTO_INSTRUMENT` (Django only) – Record subsegments for built-in database and template rendering operations.
- `AWS_XRAY_CONTEXT_MISSING` – Set to `LOG_ERROR` to avoid throwing exceptions when your instrumented code attempts to record data when no segment is open.
- `AWS_XRAY_DAEMON_ADDRESS` – Set the host and port of the X-Ray daemon listener.
- `AWS_XRAY_TRACING_NAME` – Set a service name that the SDK uses for segments.
- `PLUGINS` – Record information about your application's AWS resources.
- `SAMPLING` – Set to `False` to disable sampling.

- SAMPLING_RULES – Set the path of the JSON file containing your sampling rules.

To enable recorder configuration in settings.py, add the Django middleware to the list of installed apps.

Example settings.py – Installed apps

```
INSTALLED_APPS = [
    ...
    'django.contrib.sessions',
    'aws_xray_sdk.ext.django',
]
```

Configure the available settings in a dict named XRAY_RECORDER.

Example settings.py – Installed apps

```
XRAY_RECORDER = {
    'AUTO_INSTRUMENT': True,
    'AWS_XRAY_CONTEXT_MISSING': 'LOG_ERROR',
    'AWS_XRAY_DAEMON_ADDRESS': '127.0.0.1:5000',
    'AWS_XRAY_TRACING_NAME': 'My application',
    'PLUGINS': ('ElasticBeanstalkPlugin', 'EC2Plugin', 'ECSPlugin'),
    'SAMPLING': False,
}
```

Environment variables

You can use environment variables to configure the X-Ray SDK for Python. The SDK supports the following variables:

- AWS_XRAY_TRACING_NAME – Set a service name that the SDK uses for segments. Overrides the service name that you set programmatically.
- AWS_XRAY_SDK_ENABLED – When set to false, disables the SDK. By default, the SDK is enabled unless the environment variable is set to false.
    - When disabled, the global recorder automatically generates dummy segments and subsegments that are not sent to the daemon, and automatic patching is disabled. Middlewares are written as a wrapper over the global recorder. All segment and subsegment generation through the middleware also become dummy segment and dummy subsegments.

  - Set the value of AWS_XRAY_SDK_ENABLED through the environment variable or through direct interaction with the global_sdk_config object from the aws_xray_sdk library. Settings to the environment variable override these interactions.

- AWS_XRAY_DAEMON_ADDRESS – Set the host and port of the X-Ray daemon listener. By default, the SDK uses 127.0.0.1:2000 for both trace data (UDP) and sampling (TCP). Use this variable if you have configured the daemon to listen on a different port or if it is running on a different host.

  Format

  - **Same port** – *address*:*port*
  - **Different ports** – tcp:*address*:*port* udp:*address*:*port*

- AWS_XRAY_CONTEXT_MISSING – Set to RUNTIME_ERROR to throw exceptions when your instrumented code attempts to record data when no segment is open.

  Valid Values

  - RUNTIME_ERROR – Throw a runtime exception.
  - LOG_ERROR – Log an error and continue (default).
  - IGNORE_ERROR – Ignore error and continue.

  Errors related to missing segments or subsegments can occur when you attempt to use an instrumented client in startup code that runs when no request is open, or in code that spawns a new thread.

Environment variables override values set in code.

Tracing incoming requests with the X-Ray SDK for Python middleware

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

When you add the middleware to your application and configure a segment name, the X-Ray SDK for Python creates a segment for each sampled request. This segment includes timing, method, and disposition of the HTTP request. Additional instrumentation creates subsegments on this segment.

The X-Ray SDK for Python supports the following middleware to instrument incoming HTTP requests:

- Django
- Flask
- Bottle

Note

For AWS Lambda functions, Lambda creates a segment for each sampled request. See AWS Lambda and AWS X-Ray for more information.

See Worker for a example Python function instrumented in Lambda.

For scripts or Python applications on other frameworks, you can create segments manually.

Each segment has a name that identifies your application in the service map. The segment can be named statically, or you can configure the SDK to name it dynamically based on the host header in the incoming request. Dynamic naming lets you group traces based on the domain name in the request, and apply a default name if the name doesn't match an expected pattern (for example, if the host header is forged).

Forwarded Requests

If a load balancer or other intermediary forwards a request to your application, X-Ray takes the client IP from the `X-Forwarded-For` header in the request instead of from the source IP in the IP packet. The client IP that is recorded for a forwarded request can be forged, so it should not be trusted.

When a request is forwarded, the SDK sets an additional field in the segment to indicate this. If the segment contains the field `x_forwarded_for` set to `true`, the client IP was taken from the `X-Forwarded-For` header in the HTTP request.

The middleware creates a segment for each incoming request with an `http` block that contains the following information:

- **HTTP method –** GET, POST, PUT, DELETE, etc.
- **Client address –** The IP address of the client that sent the request.
- **Response code –** The HTTP response code for the completed request.
- **Timing –** The start time (when the request was received) and end time (when the response was sent).
- **User agent** — The `user-agent` from the request.
- **Content length** — The `content-length` from the response.

Sections

- Adding the middleware to your application (Django)
- Adding the middleware to your application (flask)
- Adding the middleware to your application (Bottle)
- Instrumenting Python code manually
- Configuring a segment naming strategy

Adding the middleware to your application (Django)

Add the middleware to the `MIDDLEWARE` list in your `settings.py` file. The X-Ray middleware should be the first line in your `settings.py` file to ensure that requests that fail in other middleware are recorded.

Example settings.py - X-Ray SDK for Python middleware

```
MIDDLEWARE = [
    'aws_xray_sdk.ext.django.middleware.XRayMiddleware',
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware'
]
```

Add the X-Ray SDK Django app to the `INSTALLED_APPS` list in your `settings.py` file. This will allow the X-Ray recorder to be configured during your app's startup.

Example settings.py - X-Ray SDK for Python Django app

```
INSTALLED_APPS = [
    'aws_xray_sdk.ext.django',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Configure a segment name in your `settings.py` file.

Example settings.py – Segment name

```
XRAY_RECORDER = {
    'AWS_XRAY_TRACING_NAME': 'My application',
    'PLUGINS': ('EC2Plugin',),
}
```

This tells the X-Ray recorder to trace requests served by your Django application with the default sampling rate. You can configure the recorder your Django settings file to apply custom sampling rules or change other settings.

> **Note**
>
> Since `plugins` are passed in as a tuple, be sure to include a trailing `,` when specifying a single plugin. For example, `plugins = ('EC2Plugin',)`

Adding the middleware to your application (flask)

To instrument your Flask application, first configure a segment name on the `xray_recorder`. Then, use the `XRayMiddleware` function to patch your Flask application in code.

Example app.py

```
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.ext.flask.middleware import XRayMiddleware

app = Flask(__name__)

xray_recorder.configure(service='My application')
XRayMiddleware(app, xray_recorder)
```

This tells the X-Ray recorder to trace requests served by your Flask application with the default sampling rate. You can configure the recorder in code to apply custom sampling rules or change other settings.

Adding the middleware to your application (Bottle)

To instrument your Bottle application, first configure a segment name on the `xray_recorder`. Then, use the `XRayMiddleware` function to patch your Bottle application in code.

Example app.py

```
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.ext.bottle.middleware import XRayMiddleware

app = Bottle()

xray_recorder.configure(service='fallback_name', dynamic_naming='My application')
app.install(XRayMiddleware(xray_recorder))
```

This tells the X-Ray recorder to trace requests served by your Bottle application with the default sampling rate. You can configure the recorder in code to apply custom sampling rules or change other settings.

Instrumenting Python code manually

If you don't use Django or Flask, you can create segments manually. You can create a segment for each incoming request, or create segments around patched HTTP or AWS SDK clients to provide context for the recorder to add subsegments.

Example main.py – Manual instrumentation

```
from aws_xray_sdk.core import xray_recorder

# Start a segment
segment = xray_recorder.begin_segment('segment_name')
# Start a subsegment
subsegment = xray_recorder.begin_subsegment('subsegment_name')

# Add metadata and annotations
segment.put_metadata('key', dict, 'namespace')
subsegment.put_annotation('key', 'value')

# Close the subsegment and segment
xray_recorder.end_subsegment()
xray_recorder.end_segment()
```

Configuring a segment naming strategy

AWS X-Ray uses a *service name* to identify your application and distinguish it from the other applications, databases, external APIs, and AWS resources that your application uses. When the X-Ray SDK generates segments for incoming requests, it records your application's service name in the segment's name field.

The X-Ray SDK can name segments after the hostname in the HTTP request header. However, this header can be forged, which could result in unexpected nodes in your service map. To prevent the SDK from naming segments incorrectly due to requests with forged host headers, you must specify a default name for incoming requests.

If your application serves requests for multiple domains, you can configure the SDK to use a dynamic naming strategy to reflect this in segment names. A dynamic naming strategy allows the SDK to use the hostname for requests that match an expected pattern, and apply the default name to requests that don't.

For example, you might have a single application serving requests to three subdomains– www.example.com, api.example.com, and static.example.com. You can use a dynamic naming strategy with the pattern *.example.com to identify segments for each subdomain with a different name, resulting in three service nodes on the service map. If your application receives requests with a hostname that doesn't match the pattern, you will see a fourth node on the service map with a fallback name that you specify.

To use the same name for all request segments, specify the name of your application when you configure the recorder, as shown in the previous sections.

A dynamic naming strategy defines a pattern that hostnames should match, and a default name to use if the hostname in the HTTP request doesn't match the pattern. To name segments dynamically in Django, add the DYNAMIC_NAMING setting to your settings.py file.

Example settings.py – Dynamic naming

```
XRAY_RECORDER = {
    'AUTO_INSTRUMENT': True,
    'AWS_XRAY_TRACING_NAME': 'My application',
    'DYNAMIC_NAMING': '*.example.com',
    'PLUGINS': ('ElasticBeanstalkPlugin', 'EC2Plugin')
}
```

You can use '*' in the pattern to match any string, or '?' to match any single character. For Flask, configure the recorder in code.

Example main.py – Segment name

```
from aws_xray_sdk.core import xray_recorder
xray_recorder.configure(service='My application')
xray_recorder.configure(dynamic_naming='*.example.com')
```

Note

You can override the default service name that you define in code with the AWS_XRAY_TRACING_NAME environment variable.

Patching libraries to instrument downstream calls

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

To instrument downstream calls, use the X-Ray SDK for Python to patch the libraries that your application uses. The X-Ray SDK for Python can patch the following libraries.

Supported Libraries

- `botocore`, `boto3` – Instrument AWS SDK for Python (Boto) clients.
- `pynamodb` – Instrument PynamoDB's version of the Amazon DynamoDB client.
- `aiobotocore`, `aioboto3` – Instrument asyncio-integrated versions of SDK for Python clients.
- `requests`, `aiohttp` – Instrument high-level HTTP clients.
- `httplib`, `http.client` – Instrument low-level HTTP clients and the higher level libraries that use them.
- `sqlite3` – Instrument SQLite clients.
- `mysql-connector-python` – Instrument MySQL clients.
- `pg8000` – Instrument Pure-Python PostgreSQL interface.
- `psycopg2` – Instrument PostgreSQL database adapter.
- `pymongo` – Instrument MongoDB clients.
- `pymysql` – Instrument PyMySQL based clients for MySQL and MariaDB.

When you use a patched library, the X-Ray SDK for Python creates a subsegment for the call and records information from the request and response. A segment must be available for the SDK to create the subsegment, either from the SDK middleware or from AWS Lambda.

Note

If you use SQLAlchemy ORM, you can instrument your SQL queries by importing the SDK's version of SQLAlchemy's session and query classes. See Use SQLAlchemy ORM for instructions.

To patch all available libraries, use the `patch_all` function in `aws_xray_sdk.core`. Some libraries, such as `httplib` and `urllib`, may need to enable double patching by calling `patch_all(double_patch=True)`.

Example main.py – Patch all supported libraries

```
import boto3
import botocore
import requests
import sqlite3

from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

patch_all()
```

To patch a single library, call `patch` with a tuple of the library name. In order to achieve this, you will need to provide a single element list.

Example main.py – Patch specific libraries

```
import boto3
import botocore
import requests
import mysql-connector-python

from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch

libraries = (['botocore'])
patch(libraries)
```

Note

In some cases, the key that you use to patch a library does not match the library name. Some keys serve as aliases for one or more libraries.

Libraries Aliases

- `httplib` – `httplib` and `http.client`
- `mysql` – `mysql-connector-python`

Tracing context for asynchronous work

For `asyncio` integrated libraries, or to create subsegments for asynchronous functions, you must also configure the X-Ray SDK for Python with an async context. Import the `AsyncContext` class and pass an instance of it to the X-Ray recorder.

Note

Web framework support libraries, such as AIOHTTP, are not handled through the `aws_xray_sdk.core.patcher` module. They will not appear in the `patcher` catalog of supported libraries.

Example main.py – Patch aioboto3

```
import asyncio
import aioboto3
import requests

from aws_xray_sdk.core.async_context import AsyncContext
from aws_xray_sdk.core import xray_recorder
xray_recorder.configure(service='my_service', context=AsyncContext())
from aws_xray_sdk.core import patch

libraries = (['aioboto3'])
patch(libraries)
```

Tracing AWS SDK calls with the X-Ray SDK for Python

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

When your application makes calls to AWS services to store data, write to a queue, or send notifications, the X-Ray SDK for Python tracks the calls downstream in subsegments. Traced AWS services and resources that you access within those services (for example, an Amazon S3 bucket or Amazon SQS queue), appear as downstream nodes on the trace map in the X-Ray console.

The X-Ray SDK for Python automatically instruments all AWS SDK clients when you patch the `botocore` library. You cannot instrument individual clients.

For all services, you can see the name of the API called in the X-Ray console. For a subset of services, the X-Ray SDK adds information to the segment to provide more granularity in the service map.

For example, when you make a call with an instrumented DynamoDB client, the SDK adds the table name to the segment for calls that target a table. In the console, each table appears as a separate node in the service map, with a generic DynamoDB node for calls that don't target a table.

Example Subsegment for a call to DynamoDB to save an item

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

When you access named resources, calls to the following services create additional nodes in the service map. Calls that don't target specific resources create a generic node for the service.

- **Amazon DynamoDB** – Table name
- **Amazon Simple Storage Service** – Bucket and key name
- **Amazon Simple Queue Service** – Queue name

Tracing calls to downstream HTTP web services using the X-Ray SDK for Python

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

When your application makes calls to microservices or public HTTP APIs, you can use the X-Ray SDK for Python to instrument those calls and add the API to the service graph as a downstream service.

To instrument HTTP clients, patch the library that you use to make outgoing calls. If you use `requests` or Python's built in HTTP client, that's all you need to do. For `aiohttp`, also configure the recorder with an async context.

If you use `aiohttp` 3's client API, you also need to configure the `ClientSession`'s with an instance of the tracing configuration provided by the SDK.

Example aiohttp 3 client API

```
from aws_xray_sdk.ext.aiohttp.client import aws_xray_trace_config

async def foo():
    trace_config = aws_xray_trace_config()
    async with ClientSession(loop=loop, trace_configs=[trace_config]) as session:
        async with session.get(url) as resp
            await resp.read()
```

When you instrument a call to a downstream web API, the X-Ray SDK for Python records a subsegment that contains information about the HTTP request and response. X-Ray uses the subsegment to generate an inferred segment for the remote API.

Example Subsegment for a downstream HTTP call

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

Example Inferred segment for a downstream HTTP call

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
  "trace_id": "1-62be1272-1b71c4274f39f122afa64eab",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  },
  "inferred": true
}
```

Generating custom subsegments with the X-Ray SDK for Python

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

Subsegments extend a trace's segment with details about work done in order to serve a request. Each time you make a call with an instrumented client, the X-Ray SDK records the information generated in a subsegment. You can create additional subsegments to group other subsegments, to measure the performance of a section of code, or to record annotations and metadata.

To manage subsegments, use the `begin_subsegment` and `end_subsegment` methods.

Example main.py – Custom subsegment

```
from aws_xray_sdk.core import xray_recorder

subsegment = xray_recorder.begin_subsegment('annotations')
subsegment.put_annotation('id', 12345)
xray_recorder.end_subsegment()
```

To create a subsegment for a synchronous function, use the `@xray_recorder.capture` decorator. You can pass a name for the subsegment to the capture function or leave it out to use the function name.

Example main.py – Function subsegment

```
from aws_xray_sdk.core import xray_recorder

@xray_recorder.capture('## create_user')
def create_user():
...
```

For an asynchronous function, use the `@xray_recorder.capture_async` decorator, and pass an async context to the recorder.

Example main.py – Asynchronous function subsegment

```
from aws_xray_sdk.core.async_context import AsyncContext
from aws_xray_sdk.core import xray_recorder
xray_recorder.configure(service='my_service', context=AsyncContext())

@xray_recorder.capture_async('## create_user')
async def create_user():
    ...

async def main():
    await myfunc()
```

When you create a subsegment within a segment or another subsegment, the X-Ray SDK for Python generates an ID for it and records the start time and end time.

Example Subsegment with metadata

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
```

Add annotations and metadata to segments with the X-Ray SDK for Python

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can record additional information about requests, the environment, or your application with annotations and metadata. You can add annotations and metadata to the segments that the X-Ray SDK creates, or to custom subsegments that you create.

Annotations are key-value pairs with string, number, or Boolean values. Annotations are indexed for use with filter expressions. Use annotations to record data that you want to use to group traces in the console, or when calling the `GetTraceSummaries` API.

Metadata are key-value pairs that can have values of any type, including objects and lists, but are not indexed for use with filter expressions. Use metadata to record additional data that you want stored in the trace but don't need to use with search.

In addition to annotations and metadata, you can also record user ID strings on segments. User IDs are recorded in a separate field on segments and are indexed for use with search.

Sections

- Recording annotations with the X-Ray SDK for Python
- Recording metadata with the X-Ray SDK for Python
- Recording user IDs with the X-Ray SDK for Python

Recording annotations with the X-Ray SDK for Python

Use annotations to record information on segments or subsegments that you want indexed for search.

Annotation Requirements

- **Keys –** The key for an X-Ray annotation can have up to 500 alphanumeric characters. You cannot use spaces or symbols other than a dot or period (.)
- **Values –** The value for an X-Ray annotation can have up to 1,000 Unicode characters.
- The number of **Annotations –** You can use up to 50 annotations per trace.

To record annotations

1. Get a reference to the current segment or subsegment from `xray_recorder`.

```
from aws_xray_sdk.core import xray_recorder
...
document = xray_recorder.current_segment()
```

or

```
from aws_xray_sdk.core import xray_recorder
...
document = xray_recorder.current_subsegment()
```

2. Call `put_annotation` with a String key, and a Boolean, Number, or String value.

```
document.put_annotation("mykey", "my value");
```

The following example shows how to call `putAnnotation` with a String key that includes a dot, and a Boolean, Number, or String value.

```
document.putAnnotation("testkey.test", "my value");
```

Alternatively, you can use the `put_annotation` method on the `xray_recorder`. This method records annotations on the current subsegment or, if no subsegment is open, on the segment.

```
xray_recorder.put_annotation("mykey", "my value");
```

The SDK records annotations as key-value pairs in an `annotations` object in the segment document. Calling `put_annotation` twice with the same key overwrites previously recorded values on the same segment or subsegment.

To find traces that have annotations with specific values, use the `annotation[key]` keyword in a filter expression.

Recording metadata with the X-Ray SDK for Python

Warning

Don't add objects with circular references as metadata values in the X-Ray SDK for Python. These objects can't be serialized into JSON and may create infinite loops in the SDK. Also, avoid adding large, complex objects as metadata to prevent performance issues.

Use metadata to record information on segments or subsegments that you don't need indexed for search. Metadata values can be strings, numbers, Booleans, or any object that can be serialized into a JSON object or array.

To record metadata

1. Get a reference to the current segment or subsegment from `xray_recorder`.

   ```
   from aws_xray_sdk.core import xray_recorder
   ...
   document = xray_recorder.current_segment()
   ```

   or

   ```
   from aws_xray_sdk.core import xray_recorder
   ...
   document = xray_recorder.current_subsegment()
   ```

2. Call `put_metadata` with a String key; a Boolean, Number, String, or Object value; and a String namespace.

```
document.put_metadata("my key", "my value", "my namespace");
```

or

Call `put_metadata` with just a key and value.

```
document.put_metadata("my key", "my value");
```

Alternatively, you can use the `put_metadata` method on the `xray_recorder`. This method records metadata on the current subsegment or, if no subsegment is open, on the segment.

```
xray_recorder.put_metadata("my key", "my value");
```

If you don't specify a namespace, the SDK uses `default`. Calling `put_metadata` twice with the same key overwrites previously recorded values on the same segment or subsegment.

Recording user IDs with the X-Ray SDK for Python

Record user IDs on request segments to identify the user who sent the request.

To record user IDs

1. Get a reference to the current segment from `xray_recorder`.

```
from aws_xray_sdk.core import xray_recorder
...
document = xray_recorder.current_segment()
```

2. Call `setUser` with a String ID of the user who sent the request.

```
document.set_user("U12345");
```

You can call `set_user` in your controllers to record the user ID as soon as your application starts processing a request.

To find traces for a user ID, use the `user` keyword in a filter expression.

Instrumenting web frameworks deployed to serverless environments

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The AWS X-Ray SDK for Python supports instrumenting web frameworks deployed in serverless applications. Serverless is the native architecture of the cloud that enables you to shift more of your operational responsibilities to AWS, increasing your agility and innovation.

Serverless architecture is a software application model that enables you to build and run applications and services without thinking about servers. It eliminates infrastructure management tasks such as server or cluster provisioning, patching, operating system maintenance, and capacity provisioning. You can build serverless solutions for nearly any type of application or backend service, and everything required to run and scale your application with high availability is handled for you.

This tutorial shows you how to automatically instrument AWS X-Ray on a web framework, such as Flask or Django, that is deployed to a serverless environment. X-Ray instrumentation of the application enables you to view all downstream calls that are made, starting from Amazon API Gateway through your AWS Lambda function, and the outgoing calls your application makes.

The X-Ray SDK for Python supports the following Python application frameworks:

- Flask version 0.8, or later
- Django version 1.0, or later

This tutorial develops an example serverless application that is deployed to Lambda and invoked by API Gateway. This tutorial uses Zappa to automatically deploy the application to Lambda and to configure the API Gateway endpoint.

Prerequisites

- Zappa
- Python – Version 2.7 or 3.6.
- AWS CLI – Verify that your AWS CLI is configured with the account and AWS Region in which you will deploy your application.
- Pip
- Virtualenv

Step 1: Create an environment

In this step, you create a virtual environment using `virtualenv` to host an application.

1. Using the AWS CLI, create a directory for the application. Then change to the new directory.

   ```
   mkdir serverless_application
   cd serverless_application
   ```

2. Next, create a virtual environment within your new directory. Use the following command to activate it.

   ```
   # Create our virtual environment
   virtualenv serverless_env

   # Activate it
   source serverless_env/bin/activate
   ```

3. Install X-Ray, Flask, Zappa, and the Requests library to your environment.

   ```
   # Install X-Ray, Flask, Zappa, and Requests into your environment
   pip install aws-xray-sdk flask zappa requests
   ```

4. Add application code to the `serverless_application` directory. For this example, we can build off of Flasks's Hello World example.

   In the `serverless_application` directory, create a file named `my_app.py`. Then use a text editor to add the following commands. This application instruments the Requests library, patches the Flask application's middleware, and opens the endpoint `'/'`.

```
# Import the X-Ray modules
from aws_xray_sdk.ext.flask.middleware import XRayMiddleware
from aws_xray_sdk.core import patcher, xray_recorder
from flask import Flask
import requests

# Patch the requests module to enable automatic instrumentation
patcher.patch(('requests',))

app = Flask(__name__)

# Configure the X-Ray recorder to generate segments with our service name
xray_recorder.configure(service='My First Serverless App')

# Instrument the Flask application
XRayMiddleware(app, xray_recorder)

@app.route('/')
def hello_world():
    resp = requests.get("https://aws.amazon.com")
    return 'Hello, World: %s' % resp.url
```

Step 2: Create and deploy a zappa environment

In this step you will use Zappa to automatically configure an API Gateway endpoint and then deploy to Lambda.

1. Initialize Zappa from within the `serverless_application` directory. For this example, we used the default settings, but if you have customization preferences, Zappa displays configuration instructions.

```
zappa init
```

```
What do you want to call this environment (default 'dev'): dev
...
What do you want to call your bucket? (default 'zappa-*******'): zappa-*******
...
...
It looks like this is a Flask application.
What's the modular path to your app's function?
```

```
This will likely be something like 'your_module.app'.
We discovered: my_app.app
Where is your app's function? (default 'my_app.app'): my_app.app
...
Would you like to deploy this application globally? (default 'n') [y/n/
(p)rimary]: n
```

2. Enable X-Ray. Open the `zappa_settings.json` file and verify that it looks similar to the example.

```
{
    "dev": {
        "app_function": "my_app.app",
        "aws_region": "us-west-2",
        "profile_name": "default",
        "project_name": "serverless-exam",
        "runtime": "python2.7",
        "s3_bucket": "zappa-*********"
    }
}
```

3. Add `"xray_tracing": true` as an entry to the configuration file.

```
{
    "dev": {
        "app_function": "my_app.app",
        "aws_region": "us-west-2",
        "profile_name": "default",
        "project_name": "serverless-exam",
        "runtime": "python2.7",
        "s3_bucket": "zappa-*********",
        "xray_tracing": true
    }
}
```

4. Deploy the application. This automatically configures the API Gateway endpoint and uploads your code to Lambda.

```
zappa deploy
```

```
...
Deploying API Gateway..
```

```
Deployment complete!: https://**********.execute-api.us-west-2.amazonaws.com/dev
```

Step 3: Enable X-Ray tracing for API Gateway

In this step you will interact with the API Gateway console to enable X-Ray tracing.

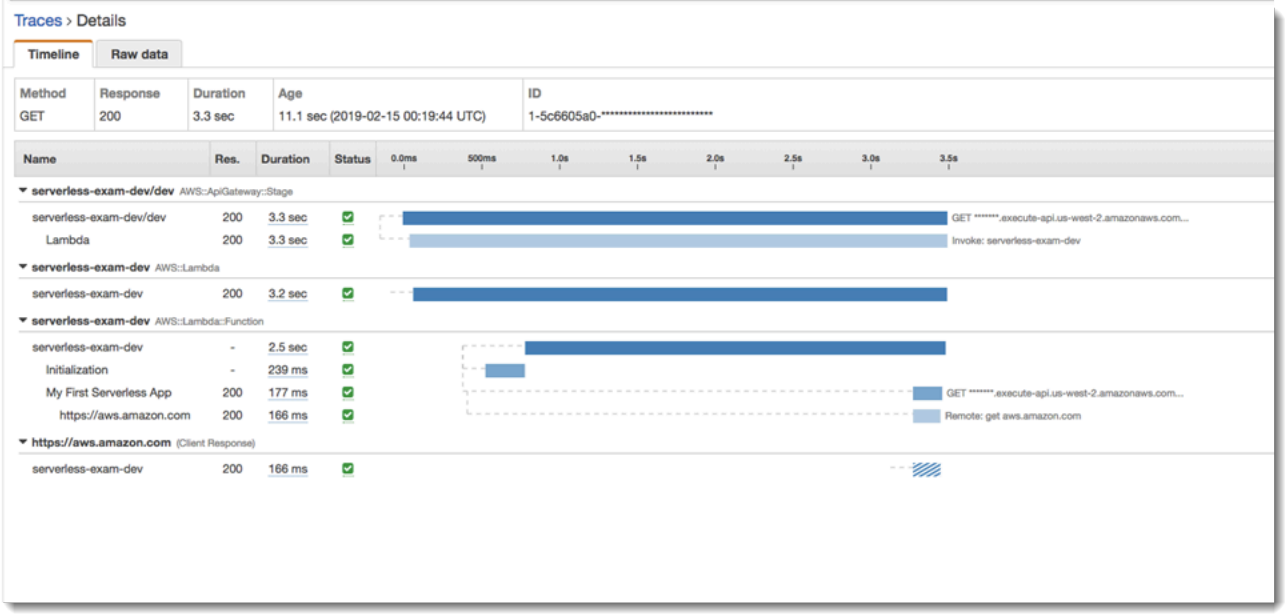
1. Sign in to the AWS Management Console and open the API Gateway console at https://console.aws.amazon.com/apigateway/.
2. Find your newly generated API. It should look something like `serverless-exam-dev`.
3. Choose **Stages**.
4. Choose the name of your deployment stage. The default is `dev`.
5. On the **Logs/Tracing** tab, select the **Enable X-Ray Tracing** box.
6. Choose **Save Changes**.
7. Access the endpoint in your browser. If you used the example `Hello World` application, it should display the following.

   ```
   "Hello, World: https://aws.amazon.com/"
   ```

Step 4: View the created trace

In this step you will interact with the X-Ray console to view the trace created by the example application. For a more detailed walkthrough on trace analysis, see Viewing the Service Map.

1. Sign in to the AWS Management Console and open the X-Ray console at https://console.aws.amazon.com/xray/home.
2. View segments generated by API Gateway, the Lambda function, and the Lambda container.
3. Under the Lambda function segment, view a subsegment named `My First Serverless App`. It's followed by a second subsegment named `https://aws.amazon.com`.
4. During initialization, Lambda might also generate a third subsegment named `initialization`.

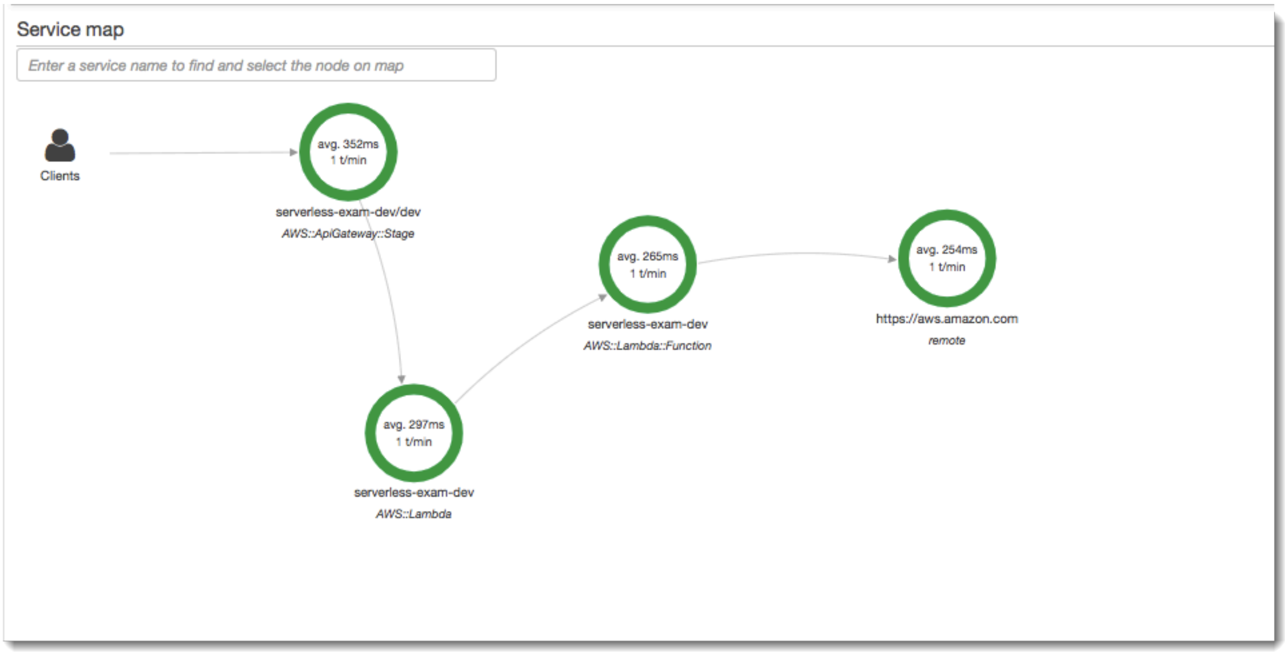


Traces › Details

Timeline | Raw data

Method	Response	Duration	Age	ID
GET	200	3.3 sec	11.1 sec (2019-02-15 00:19:44 UTC)	1-5c6605a0-*************************

Name	Res.	Duration	Status	0.0ms 500ms 1.0s 1.5s 2.0s 2.5s 3.0s 3.5s
▾ **serverless-exam-dev/dev** AWS::ApiGateway::Stage				
serverless-exam-dev/dev	200	3.3 sec	☑	GET *******.execute-api.us-west-2.amazonaws.com...
Lambda	200	3.3 sec	☑	Invoke: serverless-exam-dev
▾ **serverless-exam-dev** AWS::Lambda				
serverless-exam-dev	200	3.2 sec	☑	
▾ **serverless-exam-dev** AWS::Lambda::Function				
serverless-exam-dev	-	2.5 sec	☑	
Initialization	-	239 ms	☑	
My First Serverless App	200	177 ms	☑	GET *******.execute-api.us-west-2.amazonaws.com...
https://aws.amazon.com	200	166 ms	☑	Remote: get aws.amazon.com
▾ **https://aws.amazon.com** (Client Response)				
serverless-exam-dev	200	166 ms	☑	



Step 5: Clean up

Always terminate resources you are no longer using to avoid the accumulation of unexpected costs. As this tutorial demonstrates, tools such as Zappa streamline serverless redeployment.

To remove your application from Lambda, API Gateway, and Amazon S3, run the following command in your project directory by using the AWS CLI.

```
zappa undeploy dev
```

Next steps

Add more features to your application by adding AWS clients and instrumenting them with X-Ray. Learn more about serverless computing options at Serverless on AWS.

Working with .NET

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

There are two ways to instrument your .NET application to send traces to X-Ray:

- AWS Distro for OpenTelemetry .NET – An AWS distribution that provides a set of open source libraries for sending correlated metrics and traces to multiple AWS monitoring solutions including Amazon CloudWatch, AWS X-Ray, and Amazon OpenSearch Service, via the AWS Distro for OpenTelemetry Collector.
- AWS X-Ray SDK for .NET – A set of libraries for generating and sending traces to X-Ray via the X-Ray daemon.

For more information, see Choosing between the AWS Distro for OpenTelemetry and X-Ray SDKs.

AWS Distro for OpenTelemetry .NET

With the AWS Distro for OpenTelemetry .NET, you can instrument your applications once and send correlated metrics and traces to multiple AWS monitoring solutions including Amazon CloudWatch, AWS X-Ray, and Amazon OpenSearch Service. Using X-Ray with AWS Distro for OpenTelemetry requires two components: an *OpenTelemetry SDK* enabled for use with X-Ray, and the *AWS Distro for OpenTelemetry Collector* enabled for use with X-Ray.

To get started, see the AWS Distro for OpenTelemetry .NET documentation.

For more information about using the AWS Distro for OpenTelemetry with AWS X-Ray and other AWS services, see AWS Distro for OpenTelemetry or the AWS Distro for OpenTelemetry Documentation.

For more information about language support and usage, see AWS Observability on GitHub.

AWS X-Ray SDK for .NET

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The X-Ray SDK for .NET is a library for instrumenting C# .NET web applications, .NET Core web applications, and .NET Core functions on AWS Lambda. It provides classes and methods for generating and sending trace data to the X-Ray daemon. This includes information about incoming requests served by the application, and calls that the application makes to downstream AWS services, HTTP web APIs, and SQL databases.

> **Note**
>
> The X-Ray SDK for .NET is an open source project. You can follow the project and submit issues and pull requests on GitHub: github.com/aws/aws-xray-sdk-dotnet

For web applications, start by adding a message handler to your web configuration to trace incoming requests. The message handler creates a segment for each traced request, and completes the segment when the response is sent. While the segment is open you can use the SDK client's methods to add information to the segment and create subsegments to trace downstream calls. The SDK also automatically records exceptions that your application throws while the segment is open.

For Lambda functions called by an instrumented application or service, Lambda reads the tracing header and traces sampled requests automatically. For other functions, you can configure Lambda to sample and trace incoming requests. In either case, Lambda creates the segment and provides it to the X-Ray SDK.

Note

On Lambda, the X-Ray SDK is optional. If you don't use it in your function, your service map will still include a node for the Lambda service, and one for each Lambda function. By adding the SDK, you can instrument your function code to add subsegments to the function segment recorded by Lambda. See AWS Lambda and AWS X-Ray for more information.

Next, use the X-Ray SDK for .NET to instrument your AWS SDK for .NET clients. Whenever you make a call to a downstream AWS service or resource with an instrumented client, the SDK records information about the call in a subsegment. AWS services and the resources that you access within the services appear as downstream nodes on the trace map to help you identify errors and throttling issues on individual connections.

The X-Ray SDK for .NET also provides instrumentation for downstream calls to HTTP web APIs and SQL databases. The `GetResponseTraced` extension method for `System.Net.HttpWebRequest` traces outgoing HTTP calls. You can use the X-Ray SDK for .NET's version of `SqlCommand` to instrument SQL queries.

After you start using the SDK, customize its behavior by configuring the recorder and message handler. You can add plugins to record data about the compute resources running your application, customize sampling behavior by defining sampling rules, and set the log level to see more or less information from the SDK in your application logs.

Record additional information about requests and the work that your application does in annotations and metadata. Annotations are simple key-value pairs that are indexed for use with filter expressions, so that you can search for traces that contain specific data. Metadata entries are less restrictive and can record entire objects and arrays — anything that can be serialized into JSON.

Annotations and Metadata

Annotations and metadata are arbitrary text that you add to segments with the X-Ray SDK. Annotations are indexed for use with filter expressions. Metadata are not indexed, but can be viewed in the raw segment with the X-Ray console or API. Anyone that you grant read access to X-Ray can view this data.

When you have many instrumented clients in your code, a single request segment can contain a large number of subsegments, one for each call made with an instrumented client. You can organize and group subsegments by wrapping client calls in custom subsegments. You can create a custom subsegment for an entire function or any section of code, and record metadata and annotations on the subsegment instead of writing everything on the parent segment.

For reference documentation about the SDK's classes and methods, see the following:

- AWS X-Ray SDK for .NET API Reference
- AWS X-Ray SDK for .NET Core API Reference

The same package supports both .NET and .NET Core, but the classes that are used vary. Examples in this chapter link to the .NET API reference unless the class is specific to .NET Core.

Requirements

The X-Ray SDK for .NET requires the .NET Framework 4.5 or later and AWS SDK for .NET.

For .NET Core applications and functions, the SDK requires .NET Core 2.0 or later.

Adding the X-Ray SDK for .NET to your application

Use NuGet to add the X-Ray SDK for .NET to your application.

To install the X-Ray SDK for .NET with NuGet package manager in Visual Studio

1. Choose **Tools**, **NuGet Package Manager**, **Manage NuGet Packages for Solution**.
2. Search for **AWSXRayRecorder**.
3. Choose the package, and then choose **Install**.

Dependency management

The X-Ray SDK for .NET is available from Nuget. Install the SDK using the package manager:

```
Install-Package AWSXRayRecorder -Version 2.10.1
```

The `AWSXRayRecorder v2.10.1` nuget package has the following dependencies:

NET Framework 4.5

```
AWSXRayRecorder (2.10.1)
|
|-- AWSXRayRecorder.Core (>= 2.10.1)
|   |-- AWSSDK.Core (>= 3.3.25.1)
|
|-- AWSXRayRecorder.Handlers.AspNet (>= 2.7.3)
|   |-- AWSXRayRecorder.Core (>= 2.10.1)
|
|-- AWSXRayRecorder.Handlers.AwsSdk (>= 2.8.3)
|   |-- AWSXRayRecorder.Core (>= 2.10.1)
|
|-- AWSXRayRecorder.Handlers.EntityFramework (>= 1.1.1)
|   |-- AWSXRayRecorder.Core (>= 2.10.1)
|   |-- EntityFramework (>= 6.2.0)
|
|-- AWSXRayRecorder.Handlers.SqlServer (>= 2.7.3)
|   |-- AWSXRayRecorder.Core (>= 2.10.1)
|
|-- AWSXRayRecorder.Handlers.System.Net (>= 2.7.3)
    |-- AWSXRayRecorder.Core (>= 2.10.1)
```

NET Framework 2.0

```
AWSXRayRecorder (2.10.1)
|
|-- AWSXRayRecorder.Core (>= 2.10.1)
|   |-- AWSSDK.Core (>= 3.3.25.1)
|   |-- Microsoft.AspNetCore.Http (>= 2.0.0)
|   |-- Microsoft.Extensions.Configuration (>= 2.0.0)
|   |-- System.Net.Http (>= 4.3.4)
|
|-- AWSXRayRecorder.Handlers.AspNetCore (>= 2.7.3)
|   |-- AWSXRayRecorder.Core (>= 2.10.1)
|   |-- Microsoft.AspNetCore.Http.Extensions (>= 2.0.0)
|   |-- Microsoft.AspNetCore.Mvc.Abstractions (>= 2.0.0)
|
|-- AWSXRayRecorder.Handlers.AwsSdk (>= 2.8.3)
|   |-- AWSXRayRecorder.Core (>= 2.10.1)
```

```
|
|-- AWSXRayRecorder.Handlers.EntityFramework (>= 1.1.1)
|    |-- AWSXRayRecorder.Core (>= 2.10.1)
|    |-- Microsoft.EntityFrameworkCore.Relational (>= 3.1.0)
|
|-- AWSXRayRecorder.Handlers.SqlServer (>= 2.7.3)
|    |-- AWSXRayRecorder.Core (>= 2.10.1)
|    |-- System.Data.SqlClient (>= 4.4.0)
|
|-- AWSXRayRecorder.Handlers.System.Net (>= 2.7.3)
     |-- AWSXRayRecorder.Core (>= 2.10.1)
```

For more details about dependency management, refer to Microsoft's documentation about Nuget dependency and Nuget dependency resolution.

Configuring the X-Ray SDK for .NET

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can configure the X-Ray SDK for .NET with plugins to include information about the service that your application runs on, modify the default sampling behavior, or add sampling rules that apply to requests to specific paths.

For .NET web applications, add keys to the `appSettings` section of your `Web.config` file.

Example Web.config

```
<configuration>
  <appSettings>
    <add key="AWSXRayPlugins" value="EC2Plugin"/>
    <add key="SamplingRuleManifest" value="sampling-rules.json"/>
  </appSettings>
```

```
</configuration>
```

For .NET Core, create a file named `appsettings.json` with a top-level key named `XRay`.

Example .NET appsettings.json

```
{
  "XRay": {
    "AWSXRayPlugins": "EC2Plugin",
    "SamplingRuleManifest": "sampling-rules.json"
  }
}
```

Then, in your application code, build a configuration object and use it to initialize the X-Ray recorder. Do this before you initialize the recorder.

Example .NET Core Program.cs – Recorder configuration

```
using Amazon.XRay.Recorder.Core;
...
AWSXRayRecorder.InitializeInstance(configuration);
```

If you are instrumenting a .NET Core web application, you can also pass the configuration object to the `UseXRay` method when you configure the message handler. For Lambda functions, use the `InitializeInstance` method as shown above.

For more information on the .NET Core configuration API, see Configure an ASP.NET Core App on docs.microsoft.com.

Sections

- Plugins
- Sampling rules
- Logging (.NET)
- Logging (.NET Core)
- Environment variables

Plugins

Use plugins to add data about the service that is hosting your application.

Plugins

- Amazon EC2 – `EC2Plugin` adds the instance ID, Availability Zone, and the CloudWatch Logs Group.
- Elastic Beanstalk – `ElasticBeanstalkPlugin` adds the environment name, version label, and deployment ID.
- Amazon ECS – `ECSPlugin` adds the container ID.

To use a plugin, configure the X-Ray SDK for .NET client by adding the `AWSXRayPlugins` setting. If multiple plugins apply to your application, specify all of them in the same setting, separated by commas.

Example Web.config - plugins

```
<configuration>
  <appSettings>
    <add key="AWSXRayPlugins" value="EC2Plugin,ElasticBeanstalkPlugin"/>
  </appSettings>
</configuration>
```

Example .NET Core appsettings.json – Plugins

```
{
  "XRay": {
    "AWSXRayPlugins": "EC2Plugin,ElasticBeanstalkPlugin"
  }
}
```

Sampling rules

The SDK uses the sampling rules you define in the X-Ray console to determine which requests to record. The default rule traces the first request each second, and five percent of any additional requests across all services sending traces to X-Ray. Create additional rules in the X-Ray console to customize the amount of data recorded for each of your applications.

The SDK applies custom rules in the order in which they are defined. If a request matches multiple custom rules, the SDK applies only the first rule.

Note

If the SDK can't reach X-Ray to get sampling rules, it reverts to a default local rule of the first request each second, and five percent of any additional requests per host. This can occur if the host doesn't have permission to call sampling APIs, or can't connect to the X-Ray daemon, which acts as a TCP proxy for API calls made by the SDK.

You can also configure the SDK to load sampling rules from a JSON document. The SDK can use local rules as a backup for cases where X-Ray sampling is unavailable, or use local rules exclusively.

Example sampling-rules.json

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

This example defines one custom rule and a default rule. The custom rule applies a five-percent sampling rate with no minimum number of requests to trace for paths under `/api/move/`. The default rule traces the first request each second and 10 percent of additional requests.

The disadvantage of defining rules locally is that the fixed target is applied by each instance of the recorder independently, instead of being managed by the X-Ray service. As you deploy more hosts, the fixed rate is multiplied, making it harder to control the amount of data recorded.

On AWS Lambda, you cannot modify the sampling rate. If your function is called by an instrumented service, calls that generated requests that were sampled by that service will be recorded by Lambda. If active tracing is enabled and no tracing header is present, Lambda makes the sampling decision.

To configure backup rules, tell the X-Ray SDK for .NET to load sampling rules from a file with the `SamplingRuleManifest` setting.

Example .NET Web.config - sampling rules

```
<configuration>
  <appSettings>
    <add key="SamplingRuleManifest" value="sampling-rules.json"/>
  </appSettings>
</configuration>
```

Example .NET Core appsettings.json – Sampling rules

```
{
  "XRay": {
    "SamplingRuleManifest": "sampling-rules.json"
  }
}
```

To use only local rules, build the recorder with a `LocalizedSamplingStrategy`. If you have backup rules configured, remove that configuration.

Example .NET global.asax – Local sampling rules

```
var recorder = new AWSXRayRecorderBuilder().WithSamplingStrategy(new
 LocalizedSamplingStrategy("samplingrules.json")).Build();
AWSXRayRecorder.InitializeInstance(recorder: recorder);
```

Example .NET Core Program.cs – Local sampling rules

```
var recorder = new AWSXRayRecorderBuilder().WithSamplingStrategy(new
 LocalizedSamplingStrategy("sampling-rules.json")).Build();
AWSXRayRecorder.InitializeInstance(configuration,recorder);
```

Logging (.NET)

The X-Ray SDK for .NET uses the same logging mechanism as the AWS SDK for .NET. If you already configured your application to log AWS SDK for .NET output, the same configuration applies to output from the X-Ray SDK for .NET.

To configure logging, add a configuration section named `aws` to your `App.config` file or `Web.config` file.

Example Web.config - logging

```
...
<configuration>
  <configSections>
    <section name="aws" type="Amazon.AWSSection, AWSSDK.Core"/>
  </configSections>
  <aws>
    <logging logTo="Log4Net"/>
  </aws>
</configuration>
```

For more information, see Configuring Your AWS SDK for .NET Application in the *AWS SDK for .NET Developer Guide*.

Logging (.NET Core)

The X-Ray SDK for .NET uses the same logging options as the AWS SDK for .NET. To configure logging for .NET Core applications, pass the logging option to the `AWSXRayRecorder.RegisterLogger` method.

For example, to use log4net, create a configuration file that defines the logger, the output format, and the file location.

Example .NET Core log4net.config

```
<?xml version="1.0" encoding="utf-8" ?>
<log4net>
  <appender name="FileAppender" type="log4net.Appender.FileAppender,log4net">
    <file value="c:\logs\sdk-log.txt" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%date [%thread] %level %logger - %message%newline" />
    </layout>
  </appender>
  <logger name="Amazon">
    <level value="DEBUG" />
    <appender-ref ref="FileAppender" />
  </logger>
</log4net>
```

Then, create the logger and apply the configuration in your program code.

Example .NET Core Program.cs – Logging

```
using log4net;
using Amazon.XRay.Recorder.Core;

class Program
{
  private static ILog log;
  static Program()
  {
    var logRepository = LogManager.GetRepository(Assembly.GetEntryAssembly());
    XmlConfigurator.Configure(logRepository, new FileInfo("log4net.config"));
    log = LogManager.GetLogger(typeof(Program));
    AWSXRayRecorder.RegisterLogger(LoggingOptions.Log4Net);
  }
  static void Main(string[] args)
  {
  ...
  }
}
```

For more information on configuring log4net, see Configuration on logging.apache.org.

Environment variables

You can use environment variables to configure the X-Ray SDK for .NET. The SDK supports the following variables.

- `AWS_XRAY_TRACING_NAME` – Set a service name that the SDK uses for segments. Overrides the service name that you set on the servlet filter's segment naming strategy.
- `AWS_XRAY_DAEMON_ADDRESS` – Set the host and port of the X-Ray daemon listener. By default, the SDK uses `127.0.0.1:2000` for both trace data (UDP) and sampling (TCP). Use this variable

if you have configured the daemon to listen on a different port or if it is running on a different host.

Format

- **Same port** – *address*:*port*
- **Different ports** – tcp:*address*:*port* udp:*address*:*port*

- AWS_XRAY_CONTEXT_MISSING – Set to RUNTIME_ERROR to throw exceptions when your instrumented code attempts to record data when no segment is open.

  Valid Values

  - RUNTIME_ERROR – Throw a runtime exception.
  - LOG_ERROR – Log an error and continue (default).
  - IGNORE_ERROR – Ignore error and continue.

  Errors related to missing segments or subsegments can occur when you attempt to use an instrumented client in startup code that runs when no request is open, or in code that spawns a new thread.

Instrumenting incoming HTTP requests with the X-Ray SDK for .NET

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can use the X-Ray SDK to trace incoming HTTP requests that your application serves on an EC2 instance in Amazon EC2, AWS Elastic Beanstalk, or Amazon ECS.

Use a message handler to instrument incoming HTTP requests. When you add the X-Ray message handler to your application, the X-Ray SDK for .NET creates a segment for each sampled request. This segment includes timing, method, and disposition of the HTTP request. Additional instrumentation creates subsegments on this segment.

Note

For AWS Lambda functions, Lambda creates a segment for each sampled request. See AWS Lambda and AWS X-Ray for more information.

Each segment has a name that identifies your application in the service map. The segment can be named statically, or you can configure the SDK to name it dynamically based on the host header in the incoming request. Dynamic naming lets you group traces based on the domain name in the request, and apply a default name if the name doesn't match an expected pattern (for example, if the host header is forged).

Forwarded Requests

If a load balancer or other intermediary forwards a request to your application, X-Ray takes the client IP from the `X-Forwarded-For` header in the request instead of from the source IP in the IP packet. The client IP that is recorded for a forwarded request can be forged, so it should not be trusted.

The message handler creates a segment for each incoming request with an `http` block that contains the following information:

- **HTTP method –** GET, POST, PUT, DELETE, etc.
- **Client address –** The IP address of the client that sent the request.
- **Response code –** The HTTP response code for the completed request.
- **Timing –** The start time (when the request was received) and end time (when the response was sent).
- **User agent —** The `user-agent` from the request.
- **Content length —** The `content-length` from the response.

Sections

- Instrumenting incoming requests (.NET)

- Instrumenting incoming requests (.NET Core)
- Configuring a segment naming strategy

Instrumenting incoming requests (.NET)

To instrument requests served by your application, call `RegisterXRay` in the `Init` method of your `global.asax` file.

Example global.asax - message handler

```
using System.Web.Http;
using Amazon.XRay.Recorder.Handlers.AspNet;

namespace SampleEBWebApplication
{
  public class MvcApplication : System.Web.HttpApplication
  {
    public override void Init()
    {
      base.Init();
      AWSXRayASPNET.RegisterXRay(this, "MyApp");
    }
  }
}
```

Instrumenting incoming requests (.NET Core)

To instrument requests served by your application, call `UseXRay` method before any other middleware in the `Configure` method of your Startup class as ideally X-Ray middleware should be the first middleware to process the request and last middleware to process the response in the pipeline.

> **Note**
>
> For .NET Core 2.0, if you have a `UseExceptionHandler` method in the application, make sure to call `UseXRay` after `UseExceptionHandler` method to ensure exceptions are recorded.

Example Startup.cs

<caption>.NET Core 2.1 and above</caption>

```
using Microsoft.AspNetCore.Builder;

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    app.UseXRay("MyApp");
    // additional middleware
    ...
  }
```

<caption>.NET Core 2.0</caption>

```
using Microsoft.AspNetCore.Builder;

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    app.UseExceptionHandler("/Error");
    app.UseXRay("MyApp");
    // additional middleware
    ...
  }
```

The `UseXRay` method can also take a configuration object as a second argument.

```
app.UseXRay("MyApp", configuration);
```

Configuring a segment naming strategy

AWS X-Ray uses a *service name* to identify your application and distinguish it from the other applications, databases, external APIs, and AWS resources that your application uses. When the X-Ray SDK generates segments for incoming requests, it records your application's service name in the segment's name field.

The X-Ray SDK can name segments after the hostname in the HTTP request header. However, this header can be forged, which could result in unexpected nodes in your service map. To prevent the SDK from naming segments incorrectly due to requests with forged host headers, you must specify a default name for incoming requests.

If your application serves requests for multiple domains, you can configure the SDK to use a dynamic naming strategy to reflect this in segment names. A dynamic naming strategy allows the SDK to use the hostname for requests that match an expected pattern, and apply the default name to requests that don't.

For example, you might have a single application serving requests to three subdomains– `www.example.com`, `api.example.com`, and `static.example.com`. You can use a dynamic naming strategy with the pattern `*.example.com` to identify segments for each subdomain with a different name, resulting in three service nodes on the service map. If your application receives requests with a hostname that doesn't match the pattern, you will see a fourth node on the service map with a fallback name that you specify.

To use the same name for all request segments, specify the name of your application when you initialize the message handler, as shown in the previous section. This has the same effect as creating a `FixedSegmentNamingStrategy` and passing it to the `RegisterXRay` method.

```
AWSXRayASPNET.RegisterXRay(this, new FixedSegmentNamingStrategy("MyApp"));
```

Note

You can override the default service name that you define in code with the `AWS_XRAY_TRACING_NAME` environment variable.

A dynamic naming strategy defines a pattern that hostnames should match, and a default name to use if the hostname in the HTTP request does not match the pattern. To name segments dynamically, create a `DynamicSegmentNamingStrategy` and pass it to the `RegisterXRay` method.

```
AWSXRayASPNET.RegisterXRay(this, new DynamicSegmentNamingStrategy("MyApp",
 "*.example.com"));
```

Tracing AWS SDK calls with the X-Ray SDK for .NET

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive

updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

When your application makes calls to AWS services to store data, write to a queue, or send notifications, the X-Ray SDK for .NET tracks the calls downstream in subsegments. Traced AWS services and resources that you access within those services (for example, an Amazon S3 bucket or Amazon SQS queue), appear as downstream nodes on the trace map in the X-Ray console.

You can instrument all of your AWS SDK for .NET clients by calling `RegisterXRayForAllServices` before you create them.

Example SampleController.cs - DynamoDB client instrumentation

```
using Amazon;
using Amazon.Util;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.AwsSdk;

namespace SampleEBWebApplication.Controllers
{
  public class SampleController : ApiController
  {
    AWSSDKHandler.RegisterXRayForAllServices();
    private static readonly Lazy<AmazonDynamoDBClient> LazyDdbClient = new
 Lazy<AmazonDynamoDBClient>(() =>
    {
      var client = new AmazonDynamoDBClient(EC2InstanceMetadata.Region ??
 RegionEndpoint.USEast1);
      return client;
    });
```

To instrument clients for some services and not others, call `RegisterXRay` instead of `RegisterXRayForAllServices`. Replace the highlighted text with the name of the service's client interface.

```
AWSSDKHandler.RegisterXRay<IAmazonDynamoDB>()
```

For all services, you can see the name of the API called in the X-Ray console. For a subset of services, the X-Ray SDK adds information to the segment to provide more granularity in the service map.

For example, when you make a call with an instrumented DynamoDB client, the SDK adds the table name to the segment for calls that target a table. In the console, each table appears as a separate node in the service map, with a generic DynamoDB node for calls that don't target a table.

Example Subsegment for a call to DynamoDB to save an item

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

When you access named resources, calls to the following services create additional nodes in the service map. Calls that don't target specific resources create a generic node for the service.

- **Amazon DynamoDB** – Table name
- **Amazon Simple Storage Service** – Bucket and key name
- **Amazon Simple Queue Service** – Queue name

Tracing calls to downstream HTTP web services with the X-Ray SDK for .NET

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

When your application makes calls to microservices or public HTTP APIs, you can use the X-Ray SDK for .NET's `GetResponseTraced` extension method for `System.Net.HttpWebRequest` to instrument those calls and add the API to the service graph as a downstream service.

Example HttpWebRequest

```
using System.Net;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.System.Net;

private void MakeHttpRequest()
{
  HttpWebRequest request = (HttpWebRequest)WebRequest.Create("http://names.example.com/
api");
  request.GetResponseTraced();
}
```

For asynchronous calls, use `GetAsyncResponseTraced`.

```
request.GetAsyncResponseTraced();
```

If you use `system.net.http.httpclient`, use the `HttpClientXRayTracingHandler` delegating handler to record calls.

Example HttpClient

```
using System.Net.Http;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.System.Net;

private void MakeHttpRequest()
{
  var httpClient = new HttpClient(new HttpClientXRayTracingHandler(new
 HttpClientHandler()));
  httpClient.GetAsync(URL);
}
```

When you instrument a call to a downstream web API, the X-Ray SDK for .NET records a subsegment with information about the HTTP request and response. X-Ray uses the subsegment to generate an inferred segment for the API.

Example Subsegment for a downstream HTTP call

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

Example Inferred segment for a downstream HTTP call

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
  "trace_id": "1-62be1272-1b71c4274f39f122afa64eab",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  },
  "inferred": true
}
```

Tracing SQL queries with the X-Ray SDK for .NET

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The X-Ray SDK for .NET provides a wrapper class for `System.Data.SqlClient.SqlCommand`, named `TraceableSqlCommand`, that you can use in place of `SqlCommand`. You can initialize an SQL command with the `TraceableSqlCommand` class.

Tracing SQL queries with synchronous and asynchronous methods

The following examples show how to use the `TraceableSqlCommand` to automatically trace SQL Server queries synchronously and asynchronously.

Example `Controller.cs` - SQL client instrumentation (synchronous)

```
using Amazon;
using Amazon.Util;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.SqlServer;
```

```
private void QuerySql(int id)
{
  var connectionString = ConfigurationManager.AppSettings["RDS_CONNECTION_STRING"];
  using (var sqlConnection = new SqlConnection(connectionString))
  using (var sqlCommand = new TraceableSqlCommand("SELECT " + id, sqlConnection))
  {
    sqlCommand.Connection.Open();
    sqlCommand.ExecuteNonQuery();
  }
}
```

You can execute the query asynchronously by using the `ExecuteReaderAsync` method.

Example `Controller.cs` - SQL client instrumentation (asynchronous)

```
using Amazon;
using Amazon.Util;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.SqlServer;
private void QuerySql(int id)
{
  var connectionString = ConfigurationManager.AppSettings["RDS_CONNECTION_STRING"];
  using (var sqlConnection = new SqlConnection(connectionString))
  using (var sqlCommand = new TraceableSqlCommand("SELECT " + id, sqlConnection))
  {
    await sqlCommand.ExecuteReaderAsync();
  }
}
```

Collecting SQL queries made to SQL Server

You can enable the capture of `SqlCommand.CommandText` as part of the subsegment created by your SQL query. `SqlCommand.CommandText` appears as the field `sanitized_query` in the subsegment JSON. By default, this feature is disabled for security.

> **Note**
>
> Do not enable the collection feature if you are including sensitive information as clear text in your SQL queries.

You can enable the collection of SQL queries in two ways:

- Set the `CollectSqlQueries` property to `true` in the global configuration for your application.
- Set the `collectSqlQueries` parameter in the `TraceableSqlCommand` instance to `true` to collect calls within the instance.

Enable the global CollectSqlQueries property

The following examples show how to enable the `CollectSqlQueries` property for .NET and .NET Core.

.NET

To set the `CollectSqlQueries` property to `true` in the global configuration of your application in .NET, modify the `appsettings` of your `App.config` or `Web.config` file, as shown.

Example `App.config` Or `Web.config` – Enable SQL Query collection globally

```
<configuration>
<appSettings>
    <add key="CollectSqlQueries" value="true">
</appSettings>
</configuration>
```

.NET Core

To set the `CollectSqlQueries` property to `true` in the global configuration of your application in .NET Core, modify your `appsettings.json` file under the X-Ray key, as shown.

Example `appsettings.json` – Enable SQL Query collection globally

```
{
  "XRay": {
    "CollectSqlQueries":"true"
  }
}
```

Enable the collectSqlQueries parameter

You can set the `collectSqlQueries` parameter in the `TraceableSqlCommand` instance to `true` to collect the SQL query text for SQL Server queries made using that instance. Setting the parameter to `false` disables the `CollectSqlQuery` feature for the `TraceableSqlCommand` instance.

> **Note**
>
> The value of `collectSqlQueries` in the `TraceableSqlCommand` instance overrides the value set in the global configuration of the `CollectSqlQueries` property.

Example Example `Controller.cs` – Enable SQL Query collection for the instance

```
using Amazon;
using Amazon.Util;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.SqlServer;

private void QuerySql(int id)
{
  var connectionString = ConfigurationManager.AppSettings["RDS_CONNECTION_STRING"];
  using (var sqlConnection = new SqlConnection(connectionString))
  using (var command = new TraceableSqlCommand("SELECT " + id, sqlConnection,
 collectSqlQueries: true))
  {
    command.ExecuteNonQuery();
  }
}
```

Creating additional subsegments

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

Subsegments extend a trace's segment with details about work done in order to serve a request. Each time you make a call with an instrumented client, the X-Ray SDK records the information generated in a subsegment. You can create additional subsegments to group other subsegments, to measure the performance of a section of code, or to record annotations and metadata.

To manage subsegments, use the `BeginSubsegment` and `EndSubsegment` methods. Perform any work in the subsegment in a `try` block and use `AddException` to trace exceptions. Call `EndSubsegment` in a `finally` block to ensure that the subsegment is closed.

Example Controller.cs – Custom subsegment

```
AWSXRayRecorder.Instance.BeginSubsegment("custom method");
try
{
  DoWork();
}
catch (Exception e)
{
  AWSXRayRecorder.Instance.AddException(e);
}
finally
{
  AWSXRayRecorder.Instance.EndSubsegment();
}
```

When you create a subsegment within a segment or another subsegment, the X-Ray SDK for .NET generates an ID for it and records the start time and end time.

Example Subsegment with metadata

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
```

Add annotations and metadata to segments with the X-Ray SDK for .NET

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can record additional information about requests, the environment, or your application with annotations and metadata. You can add annotations and metadata to the segments that the X-Ray SDK creates, or to custom subsegments that you create.

Annotations are key-value pairs with string, number, or Boolean values. Annotations are indexed for use with filter expressions. Use annotations to record data that you want to use to group traces in the console, or when calling the `GetTraceSummaries` API.

Metadata are key-value pairs that can have values of any type, including objects and lists, but are not indexed for use with filter expressions. Use metadata to record additional data that you want stored in the trace but don't need to use with search.

Sections

- Recording annotations with the X-Ray SDK for .NET
- Recording metadata with the X-Ray SDK for .NET

Recording annotations with the X-Ray SDK for .NET

Use annotations to record information on segments or subsegments that you want indexed for search.

The following are required for all annotations in X-Ray:

Annotation Requirements

- **Keys –** The key for an X-Ray annotation can have up to 500 alphanumeric characters. You cannot use spaces or symbols other than a dot or period (.)
- **Values –** The value for an X-Ray annotation can have up to 1,000 Unicode characters.
- The number of **Annotations –** You can use up to 50 annotations per trace.

To record annotations outside of a AWS Lambda function

1. Get an instance of `AWSXRayRecorder`.

   ```
   using Amazon.XRay.Recorder.Core;
   ...
   AWSXRayRecorder recorder = AWSXRayRecorder.Instance;
   ```

2. Call `addAnnotation` with a String key and a Boolean, Int32, Int64, Double, or String value.

   ```
   recorder.AddAnnotation("mykey", "my value");
   ```

   The following example shows how to call `putAnnotation` with a String key that includes a dot, and a Boolean, Number, or String value.

   ```
   document.putAnnotation("testkey.test", "my value");
   ```

To record annotations inside of a AWS Lambda function

Both segments and subsegments inside a Lambda function are managed by the Lambda runtime environment. If you want to add an annotation to a segment or subsegment inside a Lambda function, you must do the following:

1. Create the segment or subsegment inside the Lambda function.
2. Add the annotation to the segment or subsegment.
3. End the segment or subsegment.

The following code example shows you how to add an annotation to a subsegment inside a Lambda function:

```
#Create the subsegment
AWSXRayRecorder.Instance.BeginSubsegment("custom method");
#Add an annotation
AWSXRayRecorder.Instance.AddAnnotation("My", "Annotation");
try
{
  YourProcess(); #Your function
}
catch (Exception e)
{
  AWSXRayRecorder.Instance.AddException(e);
}
finally #End the subsegment
{
  AWSXRayRecorder.Instance.EndSubsegment();
}
```

The X-Ray SDK records annotations as key-value pairs in an `annotations` object in the segment document. Calling the `addAnnotation` operation twice with the same key overwrites a previously recorded value on the same segment or subsegment.

To find traces that have annotations with specific values, use the `annotation[`*`key`*`]` keyword in a filter expression.

Recording metadata with the X-Ray SDK for .NET

Use metadata to record information on segments or subsegments that you don't need to index for use inside a search. Metadata values can be strings, numbers, booleans, or any other object that can be serialized into a JSON object or array.

To record metadata

1. Get an instance of `AWSXRayRecorder`, as shown in the following code example:

   ```
   using Amazon.XRay.Recorder.Core;
   ...
   AWSXRayRecorder recorder = AWSXRayRecorder.Instance;
   ```

2. Call `AddMetadata` with a string namespace, string key, and an object value, as shown in the following code example:

```
recorder.AddMetadata("my namespace", "my key", "my value");
```

You can also call the `AddMetadata` operation using just a key and value pair, as shown in the following code example:

```
recorder.AddMetadata("my key", "my value");
```

If you don't specify a value for the namespace, the X-Ray SDK uses `default`. Calling the `AddMetadata` operation twice with the same key overwrites a previously recorded value on the same segment or subsegment.

Working with Ruby

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

There are two ways to instrument your Ruby application to send traces to X-Ray:

- AWS Distro for OpenTelemetry Ruby – An AWS distribution that provides a set of open source libraries for sending correlated metrics and traces to multiple AWS monitoring solutions, including Amazon CloudWatch, AWS X-Ray, and Amazon OpenSearch Service, via the AWS Distro for OpenTelemetry Collector.
- AWS X-Ray SDK for Ruby – A set of libraries for generating and sending traces to X-Ray via the X-Ray daemon.

For more information, see Choosing between the AWS Distro for OpenTelemetry and X-Ray SDKs.

AWS Distro for OpenTelemetry Ruby

With the AWS Distro for OpenTelemetry (ADOT) Ruby, you can instrument your applications once and send correlated metrics and traces to multiple AWS monitoring solutions including Amazon CloudWatch, AWS X-Ray, and Amazon OpenSearch Service. Using X-Ray with ADOT requires two components: an *OpenTelemetry SDK* enabled for use with X-Ray, and the *AWS Distro for OpenTelemetry Collector* enabled for use with X-Ray.

To get started, see the AWS Distro for OpenTelemetry Ruby documentation.

For more information about using the AWS Distro for OpenTelemetry with AWS X-Ray and other AWS services, see AWS Distro for OpenTelemetry or the AWS Distro for OpenTelemetry Documentation.

For more information about language support and usage, see AWS Observability on GitHub.

AWS X-Ray SDK for Ruby

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The X-Ray SDK is a library for Ruby web applications that provides classes and methods for generating and sending trace data to the X-Ray daemon. Trace data includes information about incoming HTTP requests served by the application, and calls that the application makes to downstream services using the AWS SDK, HTTP clients, or an active record client. You can also create segments manually and add debug information in annotations and metadata.

You can download the SDK by adding it to your gemfile and running `bundle install`.

Example Gemfile

```
gem 'aws-sdk'
```

If you use Rails, start by adding the X-Ray SDK middleware to trace incoming requests. A request filter creates a segment. While the segment is open, you can use the SDK client's methods to add information to the segment and create subsegments to trace downstream calls. The SDK also automatically records exceptions that your application throws while the segment is open. For non-Rails applications, you can create segments manually.

Next, use the X-Ray SDK to instrument your AWS SDK for Ruby, HTTP, and SQL clients by configuring the recorder to patch the associated libraries. Whenever you make a call to a downstream AWS service or resource with an instrumented client, the SDK records information about the call in a subsegment. AWS services and the resources that you access within the services appear as downstream nodes on the trace map to help you identify errors and throttling issues on individual connections.

Once you get going with the SDK, customize its behavior by configuring the recorder. You can add plugins to record data about the compute resources running your application, customize sampling behavior by defining sampling rules, and provide a logger to see more or less information from the SDK in your application logs.

Record additional information about requests and the work that your application does in annotations and metadata. Annotations are simple key-value pairs that are indexed for use with filter expressions, so that you can search for traces that contain specific data. Metadata entries are less restrictive and can record entire objects and arrays — anything that can be serialized into JSON.

Annotations and Metadata

Annotations and metadata are arbitrary text that you add to segments with the X-Ray SDK. Annotations are indexed for use with filter expressions. Metadata are not indexed, but can be viewed in the raw segment with the X-Ray console or API. Anyone that you grant read access to X-Ray can view this data.

When you have a lot of instrumented clients in your code, a single request segment can contain a large number of subsegments, one for each call made with an instrumented client. You can organize and group subsegments by wrapping client calls in custom subsegments. You can create a custom subsegment for an entire function or any section of code, and record metadata and annotations on the subsegment instead of writing everything on the parent segment.

For reference documentation for the SDK's classes and methods, see the AWS X-Ray SDK for Ruby API Reference.

Requirements

The X-Ray SDK requires Ruby 2.3 or later and is compatible with the following libraries:

- AWS SDK for Ruby version 3.0 or later
- Rails version 5.1 or later

Configuring the X-Ray SDK for Ruby

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

The X-Ray SDK for Ruby has a class named `XRay.recorder` that provides the global recorder. You can configure the global recorder to customize the middleware that creates segments for incoming HTTP calls.

Sections

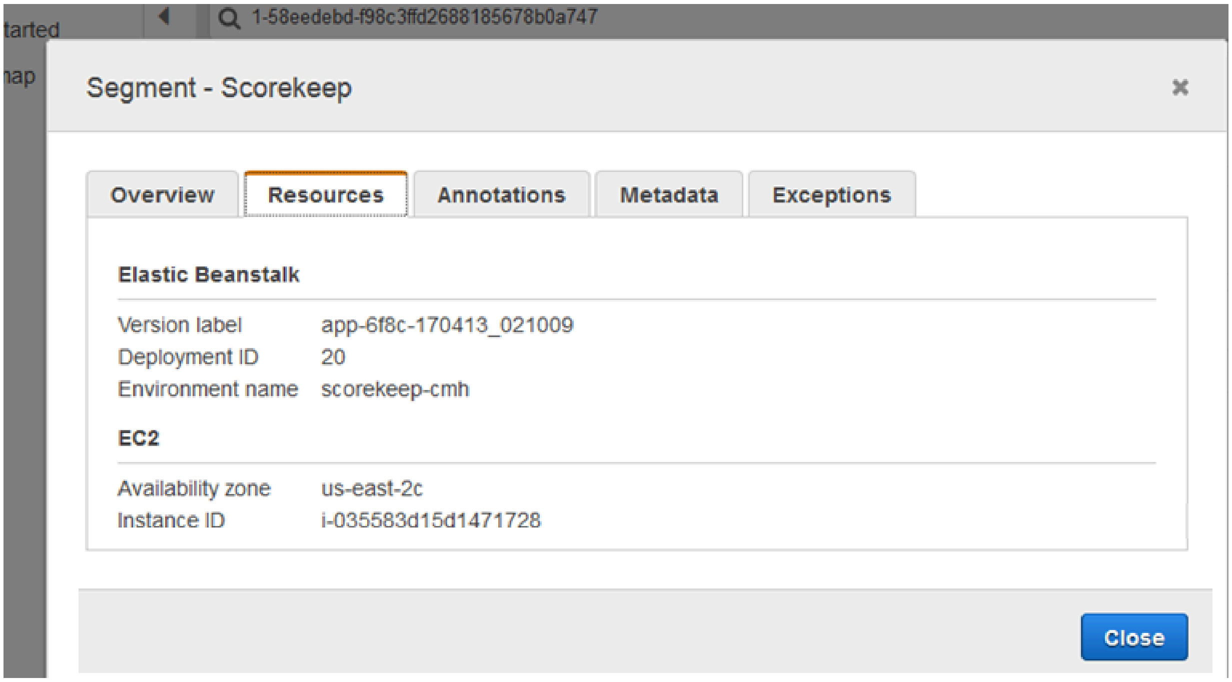
- Service plugins
- Sampling rules
- Logging
- Recorder configuration in code
- Recorder configuration with rails
- Environment variables

Service plugins

Use `plugins` to record information about the service hosting your application.

Plugins

- Amazon EC2 – `ec2` adds the instance ID and Availability Zone.
- Elastic Beanstalk – `elastic_beanstalk` adds the environment name, version label, and deployment ID.
- Amazon ECS – `ecs` adds the container ID.



Segment - Scorekeep

Elastic Beanstalk

Version label	app-6f8c-170413_021009
Deployment ID	20
Environment name	scorekeep-cmh

EC2

Availability zone	us-east-2c
Instance ID	i-035583d15d1471728

To use plugins, specify it in the configuration object that you pass to the recorder.

Example main.rb – Plugin configuration

```
my_plugins = %I[ec2 elastic_beanstalk]

config = {
  plugins: my_plugins,
  name: 'my app',
}

XRay.recorder.configure(config)
```

You can also use environment variables, which take precedence over values set in code, to configure the recorder.

The SDK also uses plugin settings to set the `origin` field on the segment. This indicates the type of AWS resource that runs your application. When you use multiple plugins, the SDK uses the following resolution order to determine the origin: ElasticBeanstalk > EKS > ECS > EC2.

Sampling rules

The SDK uses the sampling rules you define in the X-Ray console to determine which requests to record. The default rule traces the first request each second, and five percent of any additional requests across all services sending traces to X-Ray. Create additional rules in the X-Ray console to customize the amount of data recorded for each of your applications.

The SDK applies custom rules in the order in which they are defined. If a request matches multiple custom rules, the SDK applies only the first rule.

> **Note**
>
> If the SDK can't reach X-Ray to get sampling rules, it reverts to a default local rule of the first request each second, and five percent of any additional requests per host. This can occur if the host doesn't have permission to call sampling APIs, or can't connect to the X-Ray daemon, which acts as a TCP proxy for API calls made by the SDK.

You can also configure the SDK to load sampling rules from a JSON document. The SDK can use local rules as a backup for cases where X-Ray sampling is unavailable, or use local rules exclusively.

Example sampling-rules.json

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

This example defines one custom rule and a default rule. The custom rule applies a five-percent sampling rate with no minimum number of requests to trace for paths under `/api/move/`. The default rule traces the first request each second and 10 percent of additional requests.

The disadvantage of defining rules locally is that the fixed target is applied by each instance of the recorder independently, instead of being managed by the X-Ray service. As you deploy more hosts, the fixed rate is multiplied, making it harder to control the amount of data recorded.

To configure backup rules, define a hash for the document in the configuration object that you pass to the recorder.

Example main.rb – Backup rule configuration

```
require 'aws-xray-sdk'
my_sampling_rules =  {
  version: 1,
  default: {
    fixed_target: 1,
    rate: 0.1
  }
}
config = {
  sampling_rules: my_sampling_rules,
  name: 'my app',
}
XRay.recorder.configure(config)
```

To store the sampling rules independently, define the hash in a separate file and require the file to pull it into your application.

Example config/sampling-rules.rb

```
my_sampling_rules =  {
  version: 1,
  default: {
    fixed_target: 1,
    rate: 0.1
  }
}
```

Example main.rb – Sampling rule from a file

```
require 'aws-xray-sdk'
require 'config/sampling-rules.rb'

config = {
  sampling_rules: my_sampling_rules,
  name: 'my app',
}
XRay.recorder.configure(config)
```

To use only local rules, require the sampling rules and configure the `LocalSampler`.

Example main.rb – Local rule sampling

```
require 'aws-xray-sdk'
require 'aws-xray-sdk/sampling/local/sampler'

config = {
  sampler: LocalSampler.new,
  name: 'my app',
}
XRay.recorder.configure(config)
```

You can also configure the global recorder to disable sampling and instrument all incoming requests.

Example main.rb – Disable sampling

```
require 'aws-xray-sdk'
config = {
  sampling: false,
  name: 'my app',
}
XRay.recorder.configure(config)
```

Logging

By default, the recorder outputs info-level events to `$stdout`. You can customize logging by defining a logger in the configuration object that you pass to the recorder.

Example main.rb – Logging

```
require 'aws-xray-sdk'
config = {
  logger: my_logger,
  name: 'my app',
}
```

```
XRay.recorder.configure(config)
```

Use debug logs to identify issues, such as unclosed subsegments, when you generate subsegments manually.

Recorder configuration in code

Additional settings are available from the `configure` method on `XRay.recorder`.

- `context_missing` – Set to `LOG_ERROR` to avoid throwing exceptions when your instrumented code attempts to record data when no segment is open.
- `daemon_address` – Set the host and port of the X-Ray daemon listener.
- `name` – Set a service name that the SDK uses for segments.
- `naming_pattern` – Set a domain name pattern to use dynamic naming.
- `plugins` – Record information about your application's AWS resources with plugins.
- `sampling` – Set to `false` to disable sampling.
- `sampling_rules` – Set the hash containing your sampling rules.

Example main.rb – Disable context missing exceptions

```
require 'aws-xray-sdk'
config = {
  context_missing: 'LOG_ERROR'
}

XRay.recorder.configure(config)
```

Recorder configuration with rails

If you use the Rails framework, you can configure options on the global recorder in a Ruby file under `app_root/initializers`. The X-Ray SDK supports an additional configuration key for use with Rails.

- `active_record` – Set to `true` to record subsegments for Active Record database transactions.

Configure the available settings in a configuration object named `Rails.application.config.xray`.

Example config/initializers/aws_xray.rb

```
Rails.application.config.xray = {
  name: 'my app',
  patch: %I[net_http aws_sdk],
  active_record: true
}
```

Environment variables

You can use environment variables to configure the X-Ray SDK for Ruby. The SDK supports the following variables:

- `AWS_XRAY_TRACING_NAME` – Set a service name that the SDK uses for segments. Overrides the service name that you set on the servlet filter's segment naming strategy.
- `AWS_XRAY_DAEMON_ADDRESS` – Set the host and port of the X-Ray daemon listener. By default, the SDK sends trace data to `127.0.0.1:2000`. Use this variable if you have configured the daemon to listen on a different port or if it is running on a different host.
- `AWS_XRAY_CONTEXT_MISSING` – Set to `RUNTIME_ERROR` to throw exceptions when your instrumented code attempts to record data when no segment is open.

  Valid Values

  - `RUNTIME_ERROR` – Throw a runtime exception.
  - `LOG_ERROR` – Log an error and continue (default).
  - `IGNORE_ERROR` – Ignore error and continue.

  Errors related to missing segments or subsegments can occur when you attempt to use an instrumented client in startup code that runs when no request is open, or in code that spawns a new thread.

Environment variables override values set in code.

Tracing incoming requests with the X-Ray SDK for Ruby middleware

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive

updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can use the X-Ray SDK to trace incoming HTTP requests that your application serves on an EC2 instance in Amazon EC2, AWS Elastic Beanstalk, or Amazon ECS.

If you use Rails, use the Rails middleware to instrument incoming HTTP requests. When you add the middleware to your application and configure a segment name, the X-Ray SDK for Ruby creates a segment for each sampled request. Any segments created by additional instrumentation become subsegments of the request-level segment that provides information about the HTTP request and response. This information includes timing, method, and disposition of the request.

Each segment has a name that identifies your application in the service map. The segment can be named statically, or you can configure the SDK to name it dynamically based on the host header in the incoming request. Dynamic naming lets you group traces based on the domain name in the request, and apply a default name if the name doesn't match an expected pattern (for example, if the host header is forged).

Forwarded Requests

If a load balancer or other intermediary forwards a request to your application, X-Ray takes the client IP from the `X-Forwarded-For` header in the request instead of from the source IP in the IP packet. The client IP that is recorded for a forwarded request can be forged, so it should not be trusted.

When a request is forwarded, the SDK sets an additional field in the segment to indicate this. If the segment contains the field `x_forwarded_for` set to `true`, the client IP was taken from the `X-Forwarded-For` header in the HTTP request.

The middleware creates a segment for each incoming request with an `http` block that contains the following information:

- **HTTP method –** GET, POST, PUT, DELETE, etc.
- **Client address –** The IP address of the client that sent the request.

- **Response code** – The HTTP response code for the completed request.
- **Timing** – The start time (when the request was received) and end time (when the response was sent).
- **User agent** — The `user-agent` from the request.
- **Content length** — The `content-length` from the response.

Using the rails middleware

To use the middleware, update your gemfile to include the required railtie.

Example Gemfile - rails

```
gem 'aws-xray-sdk', require: ['aws-xray-sdk/facets/rails/railtie']
```

To use the middleware, you must also configure the recorder with a name that represents the application in the trace map.

Example config/initializers/aws_xray.rb

```
Rails.application.config.xray = {
  name: 'my app'
}
```

Instrumenting code manually

If you don't use Rails, create segments manually. You can create a segment for each incoming request, or create segments around patched HTTP or AWS SDK clients to provide context for the recorder to add subsegments.

```
# Start a segment
segment = XRay.recorder.begin_segment 'my_service'
# Start a subsegment
subsegment = XRay.recorder.begin_subsegment 'outbound_call', namespace: 'remote'

# Add metadata or annotation here if necessary
my_annotations = {
  k1: 'v1',
  k2: 1024
}
segment.annotations.update my_annotations
```

```
# Add metadata to default namespace
subsegment.metadata[:k1] = 'v1'

# Set user for the segment (subsegment is not supported)
segment.user = 'my_name'

# End segment/subsegment
XRay.recorder.end_subsegment
XRay.recorder.end_segment
```

Configuring a segment naming strategy

AWS X-Ray uses a *service name* to identify your application and distinguish it from the other applications, databases, external APIs, and AWS resources that your application uses. When the X-Ray SDK generates segments for incoming requests, it records your application's service name in the segment's name field.

The X-Ray SDK can name segments after the hostname in the HTTP request header. However, this header can be forged, which could result in unexpected nodes in your service map. To prevent the SDK from naming segments incorrectly due to requests with forged host headers, you must specify a default name for incoming requests.

If your application serves requests for multiple domains, you can configure the SDK to use a dynamic naming strategy to reflect this in segment names. A dynamic naming strategy allows the SDK to use the hostname for requests that match an expected pattern, and apply the default name to requests that don't.

For example, you might have a single application serving requests to three subdomains– `www.example.com`, `api.example.com`, and `static.example.com`. You can use a dynamic naming strategy with the pattern `*.example.com` to identify segments for each subdomain with a different name, resulting in three service nodes on the service map. If your application receives requests with a hostname that doesn't match the pattern, you will see a fourth node on the service map with a fallback name that you specify.

To use the same name for all request segments, specify the name of your application when you configure the recorder, as shown in the previous sections.

A dynamic naming strategy defines a pattern that hostnames should match, and a default name to use if the hostname in the HTTP request doesn't match the pattern. To name segments dynamically, specify a naming pattern in the config hash.

Example main.rb – Dynamic naming

```
config = {
  naming_pattern: '*mydomain*',
  name: 'my app',
}

XRay.recorder.configure(config)
```

You can use '*' in the pattern to match any string, or '?' to match any single character.

Note

You can override the default service name that you define in code with the `AWS_XRAY_TRACING_NAME` environment variable.

Patching libraries to instrument downstream calls

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

To instrument downstream calls, use the X-Ray SDK for Ruby to patch the libraries that your application uses. The X-Ray SDK for Ruby can patch the following libraries.

Supported Libraries

- `net/http` – Instrument HTTP clients.
- `aws-sdk` – Instrument AWS SDK for Ruby clients.

When you use a patched library, the X-Ray SDK for Ruby creates a subsegment for the call and records information from the request and response. A segment must be available for the SDK to create the subsegment, either from the SDK middleware or a call to `XRay.recorder.begin_segment`.

To patch libraries, specify them in the configuration object that you pass to the X-Ray recorder.

Example main.rb – Patch libraries

```
require 'aws-xray-sdk'

config = {
  name: 'my app',
  patch: %I[net_http aws_sdk]
}

XRay.recorder.configure(config)
```

Tracing AWS SDK calls with the X-Ray SDK for Ruby

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

When your application makes calls to AWS services to store data, write to a queue, or send notifications, the X-Ray SDK for Ruby tracks the calls downstream in subsegments. Traced AWS services and resources that you access within those services (for example, an Amazon S3 bucket or Amazon SQS queue), appear as downstream nodes on the trace map in the X-Ray console.

The X-Ray SDK for Ruby automatically instruments all AWS SDK clients when you patch the aws-sdk library. You cannot instrument individual clients.

For all services, you can see the name of the API called in the X-Ray console. For a subset of services, the X-Ray SDK adds information to the segment to provide more granularity in the service map.

For example, when you make a call with an instrumented DynamoDB client, the SDK adds the table name to the segment for calls that target a table. In the console, each table appears as a separate node in the service map, with a generic DynamoDB node for calls that don't target a table.

Example Subsegment for a call to DynamoDB to save an item

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

When you access named resources, calls to the following services create additional nodes in the service map. Calls that don't target specific resources create a generic node for the service.

- **Amazon DynamoDB –** Table name
- **Amazon Simple Storage Service –** Bucket and key name
- **Amazon Simple Queue Service –** Queue name

Generating custom subsegments with the X-Ray SDK

Note

End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

Subsegments extend a trace's segment with details about work done in order to serve a request. Each time you make a call with an instrumented client, the X-Ray SDK records the information generated in a subsegment. You can create additional subsegments to group other subsegments, to measure the performance of a section of code, or to record annotations and metadata.

To manage subsegments, use the `begin_subsegment` and `end_subsegment` methods.

```
subsegment = XRay.recorder.begin_subsegment name: 'annotations', namespace: 'remote'
my_annotations = { id: 12345 }
subsegment.annotations.update my_annotations
XRay.recorder.end_subsegment
```

To create a subsegment for a function, wrap it in a call to `XRay.recorder.capture`.

```
XRay.recorder.capture('name_for_subsegment') do |subsegment|
  resp = myfunc() # myfunc is your function
  subsegment.annotations.update k1: 'v1'
  resp
end
```

When you create a subsegment within a segment or another subsegment, the X-Ray SDK generates an ID for it and records the start time and end time.

Example Subsegment with metadata

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
```

Add annotations and metadata to segments with the X-Ray SDK for Ruby

> **Note**
>
> End-of-support notice – On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. After February 25th, 2027, you will no longer receive updates or releases. For more information on the support timeline, see X-Ray SDK and daemon end of support timeline. We recommend to migrate to OpenTelemetry. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

You can record additional information about requests, the environment, or your application with annotations and metadata. You can add annotations and metadata to the segments that the X-Ray SDK creates, or to custom subsegments that you create.

Annotations are key-value pairs with string, number, or Boolean values. Annotations are indexed for use with filter expressions. Use annotations to record data that you want to use to group traces in the console, or when calling the `GetTraceSummaries` API.

Metadata are key-value pairs that can have values of any type, including objects and lists, but are not indexed for use with filter expressions. Use metadata to record additional data that you want stored in the trace but don't need to use with search.

In addition to annotations and metadata, you can also record user ID strings on segments. User IDs are recorded in a separate field on segments and are indexed for use with search.

Sections

- Recording annotations with the X-Ray SDK for Ruby
- Recording metadata with the X-Ray SDK for Ruby

- [Recording user IDs with the X-Ray SDK for Ruby]

Recording annotations with the X-Ray SDK for Ruby

Use annotations to record information on segments or subsegments that you want indexed for search.

Annotation Requirements

- **Keys –** The key for an X-Ray annotation can have up to 500 alphanumeric characters. You cannot use spaces or symbols other than a dot or period (.)
- **Values –** The value for an X-Ray annotation can have up to 1,000 Unicode characters.
- The number of **Annotations –** You can use up to 50 annotations per trace.

To record annotations

1. Get a reference to the current segment or subsegment from `xray_recorder`.

```
require 'aws-xray-sdk'
...
document = XRay.recorder.current_segment
```

   or

```
require 'aws-xray-sdk'
...
document = XRay.recorder.current_subsegment
```

2. Call `update` with a hash value.

```
my_annotations = { id: 12345 }
document.annotations.update my_annotations
```

   The following is an example that shows how to call `update` with an annotation key that contains a dot.

```
my_annotations = { testkey.test: 12345 }
document.annotations.update my_annotations
```

The SDK records annotations as key-value pairs in an annotations object in the segment document. Calling add_annotations twice with the same key overwrites previously recorded values on the same segment or subsegment.

To find traces that have annotations with specific values, use the annotation[*key*] keyword in a filter expression.

Recording metadata with the X-Ray SDK for Ruby

Use metadata to record information on segments or subsegments that you don't need indexed for search. Metadata values can be strings, numbers, Booleans, or any object that can be serialized into a JSON object or array.

To record metadata

1. Get a reference to the current segment or subsegment from xray_recorder.

```
require 'aws-xray-sdk'
...
document = XRay.recorder.current_segment
```

   or

```
require 'aws-xray-sdk'
...
document = XRay.recorder.current_subsegment
```

2. Call metadata with a String key; a Boolean, Number, String, or Object value; and a String namespace.

```
my_metadata = {
  my_namespace: {
    key: 'value'
  }
}
subsegment.metadata my_metadata
```

Calling metadata twice with the same key overwrites previously recorded values on the same segment or subsegment.

Recording user IDs with the X-Ray SDK for Ruby

Record user IDs on request segments to identify the user who sent the request.

To record user IDs

1. Get a reference to the current segment from `xray_recorder`.

   ```
   require 'aws-xray-sdk'
   ...
   document = XRay.recorder.current_segment
   ```

2. Set the user field on the segment to a String ID of the user who sent the request.

   ```
   segment.user = 'U12345'
   ```

You can set the user in your controllers to record the user ID as soon as your application starts processing a request.

To find traces for a user ID, use the `user` keyword in a filter expression.

X-Ray SDK and daemon end of support timeline

The following table lists the dates and the level of support for X-Ray SDKs and daemon.

SDK and daemon phase	Start date	End date	Support provided
General availability	NA	February 25th, 2026	X-Ray SDKs and daemon are fully supported. AWS provides regular SDK and daemon releases that include bug and security fixes.
Maintenance mode	February 25th, 2026	February 25th, 2027	AWS will limit X-Ray SDK and daemon releases to address only critical security issues. No new feature enhanceme nts.
End of support	February 25th, 2027	NA	X-Ray SDKs and daemon will no longer receive updates or releases. All published releases will continue to be available through public package managers and the code will remain on GitHub.

Before February 25th, 2027 we recommend that you migrate to OpenTelemetry solutions for instrumenting your application and sending traces to AWS X-Ray. For more information on migrating to OpenTelemetry, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation .

Migrating from X-Ray instrumentation to OpenTelemetry instrumentation

Note

The AWS X-Ray SDKs and daemon enter maintenance mode on February 25th, 2026 and reach end of support on February 25th, 2027. After February 25th, 2027, you will no longer receive updates or releases. For more information on the dates and the level of support for X-Ray SDKs and daemon, see X-Ray SDK and daemon end of support timeline.

X-Ray is transitioning to OpenTelemetry (OTel) as its primary instrumentation standard for application tracing and observability. This strategic shift aligns AWS with industry best practices and offers customers a more comprehensive, flexible, and future-ready solution for their observability needs. OpenTelemetry's wide adoption in the industry enables tracing of requests across diverse systems, including those outside AWS that may not directly integrate with X-Ray.

This chapter provides recommendations for a smooth transition, and emphasizes the importance of migrating to OpenTelemetry-based solutions to ensure continued support and access to the latest features in application instrumentation and observability.

It is recommended to adopt OpenTelemetry as the observability solution for instrumenting your application.

Topics

- Understanding OpenTelemetry
- Understanding OpenTelemetry concepts for migration
- Migration overview
- Migrating from X-Ray Daemon to AWS CloudWatch agent or OpenTelemetry collector
- Migrating to OpenTelemetry Java
- Migrate to OpenTelemetry Go
- Migrate to OpenTelemetry Node.js
- Migrate to OpenTelemetry .NET
- Migrate to OpenTelemetry Python

- Migrate to OpenTelemetry Ruby

Understanding OpenTelemetry

OpenTelemetry is an industry-standard observability framework that provides standardized protocols and tools for collecting telemetry data. It offers a unified approach to instrumenting, generating, collecting, and exporting telemetry data such as metrics, logs, and traces.

When you migrate from X-Ray SDKs to OpenTelemetry, you get the following benefits:

- Enhanced framework and library instrumentation support
- Support for additional programming languages
- Automatic instrumentation capabilities
- Flexible sampling configuration options
- Unified collection of metrics, logs, and traces

The OpenTelemetry collector provides more options for data collection formats and export destinations than the X-Ray daemon.

OpenTelemetry support in AWS

AWS provides multiple solutions for working with OpenTelemetry:

- AWS Distro for OpenTelemetry

  Export OpenTelemetry traces as segments to X-Ray.

  For more information, see AWS Distro for OpenTelemetry.
- CloudWatch Application Signals

  Export customized OpenTelemetry traces and metrics to monitor application health.

  For more information, see Working with Application Signals.
- CloudWatch OTel Endpoint

  Export OpenTelemetry traces to X-Ray using the HTTP OTel endpoint with native OpenTelemetry instrumentation.

For more information, see Using OTel endpoints.

Using OpenTelemetry with AWS CloudWatch

AWS CloudWatch supports OpenTelemetry traces through client-side application instrumentation and native AWS CloudWatch services such as Application Signals, Trace, Map, Metrics, and Logs. For more information, see OpenTelemetry.

Understanding OpenTelemetry concepts for migration

The following table maps X-Ray concepts to their OpenTelemetry equivalents. Understanding these mappings helps you translate your existing X-Ray instrumentation to OpenTelemetry:

X-Ray concept	OpenTelemetry concept
X-Ray Recorder	Tracer Provider and Tracers
Service Plugins	Resource Detector
Segment	(Server) Span
Sub-segment	(non-Server) Span
X-Ray Sampling Rules	OpenTelemetry Sampling (Customizable)
X-Ray Emitter	Span Exporter (Customizable)
Annotations/Metadata	Attributes
Library Instrumentation	Library Instrumentation
X-Ray Trace Context	Span Context
X-Ray Trace Context Propagation	W3C Trace Context Propagation
X-Ray Trace Sampling	OpenTelemetry Trace Sampling
N/A	Span Processing

X-Ray concept	OpenTelemetry concept
N/A	Baggage
X-Ray Daemon	OpenTelemetry Collector

Note

For more information about OpenTelemetry concepts, see the OpenTelemetry documentation.

Comparing features

The following table shows which features are supported in both services. Use this information to identify any gaps you need to address during migration:

Feature	X-Ray instrumentation	OpenTelemetry instrumentation
Library instrumentation	Supported	Supported
X-Ray sampling	Supported	Supported in OTel Java/.NET/Go Supported in ADOT Java/.NET/Python/Node.js
X-Ray trace context propagation	Supported	Supported
Resource detection	Supported	Supported
Segment annotations	Supported	Supported
Segment metadata	Supported	Supported
Zero-code auto-instrumentation	Supported in Java	Supported in OTel Java/.NET/Python/Node.js

Feature	X-Ray instrumentation	OpenTelemetry instrumentation
		Supported in ADOT Java/.NET /Python/Node.js
Manually trace creation	Supported	Supported

Setting up and configuring tracing

To create traces in OpenTelemetry, you need a tracer. You get a tracer by initializing a *Tracer Provider* in your application. This is similar to how you use the X-Ray Recorder to configure X-Ray and create segments and subsegments in an X-Ray trace.

> **Note**
>
> The OpenTelemetry *Tracer Provider* offers more configuration options than the X-Ray Recorder.

Understanding trace data structure

After understanding the basic concepts and feature mappings, you can learn about specific implementation details like trace data structure and sampling.

OpenTelemetry uses *spans* instead of segments and subsegments to structure trace data. Each span includes the following components:

- Name
- Unique ID
- Start and end timestamps
- Span kind
- Span context
- Attributes (key-value metadata)
- Events (timestamped logs)
- Links to other spans

- Status information
- Parent span references

When you migrate to OpenTelemetry, your spans are automatically converted to X-Ray segments or subsegments. This ensures your existing CloudWatch console experience remains unchanged.

Working with span attributes

The X-Ray SDK provides two ways to add data to segments and subsegments:

Annotations

Key-value pairs that are indexed for filtering and searching

Metadata

Key-value pairs containing complex data that isn't indexed for searching

By default, OpenTelemetry span attributes are converted to metadata in X-Ray raw data. To convert specific attributes to annotations instead, add their keys to the `aws.xray.annotations` attributes list.

- For more information about OpenTelemetry concepts, see OpenTelemetry Traces
- For details about how OpenTelemetry data maps to X-Ray data, see OpenTelemetry to X-Ray data model translation

Detecting resources in your environment

OpenTelemetry uses *Resource Detectors* to collect metadata about the resources that generate telemetry data. This metadata is stored as *Resource Attributes*. For example, an entity producing telemetry could be an Amazon ECS cluster or an Amazon EC2 instance, and the Resource Attributes that can be recorded from these entities can include the Amazon ECS Cluster ARN or Amazon EC2 Instance ID.

- For information about supported resource types, see OpenTelemetry Resource Semantic Conventions
- For information about X-Ray service plugins, see Configuring the X-Ray SDK

Managing sampling strategies

Trace sampling helps you manage costs by collecting data from a representative subset of requests instead of all requests. Both OpenTelemetry and X-Ray support sampling, but implement it differently.

> **Note**
>
> Sampling fewer than 100% of traces reduces your observability costs while maintaining meaningful insights into your application's performance.

OpenTelemetry provides several built-in sampling strategies and lets you create custom ones. You can also configure an X-Ray *Remote Sampler* in some SDK languages to use X-Ray sampling rules with OpenTelemetry.

The additional sampling strategies from OpenTelemetry are:

- Parent-based Sampling – Respects the parent span's sampling decision before applying additional sampling strategies
- Trace ID Ratio Based Sampling – >Randomly samples a specified percentage of spans
- Tail sampling – Applies sampling rules to complete traces in the OpenTelemetry Collector
- Custom samplers – Implement your own sampling logic using the sampling interface

For information about X-Ray sampling rules, see Sampling rules in the X-Ray console

For information about OpenTelemetry tail sampling, see Tail sampling processor

Managing trace context

X-Ray SDKs manage the Segment Context to correctly handle parent-child relationships between Segments and Subsegments in a trace. OpenTelemetry uses a similar mechanism to ensure that spans have the correct parent span. It stores and propagates tracing data throughout a request context. For example, when your application processes a request and creates a server span to represent that request, OpenTelemetry will store the server span in the OpenTelemetry Context so that when a child span is created, that child span can reference the span in the Context as its parent.

Propagating trace context

Both X-Ray and OpenTelemetry use HTTP headers to propagate trace context across services. This allows you to link trace data generated by different services and maintain sampling decisions.

The X-Ray SDK automatically propagates trace context using the X-Ray trace header. When one service calls another, the trace header contains the context needed to maintain parent-child relationships between traces.

OpenTelemetry supports multiple trace header formats for context propagation, including:

- W3C Trace Context (default)
- X-Ray trace header
- Other custom formats

> **Note**
>
> You can configure OpenTelemetry to use one or more header formats. For example, use the X-Ray Propagator to send trace context to AWS services that support X-Ray tracing.

Configure and use the X-Ray Propagator to enable tracing across AWS services. This allows you to propagate trace context to API Gateway endpoints and other services that support X-Ray.

- For information about X-Ray trace headers, see Tracing header in the X-Ray Developer Guide
- For information about OpenTelemetry context propagation, see Context and Context Propagation in the OpenTelemetry documentation

Using library instrumentation

Both X-Ray and OpenTelemetry provide library instrumentation that requires minimal code changes to add tracing to your applications.

X-Ray provides library instrumentation functionalities. This allows you to add pre-built X-Ray instrumentations with minimal application code changes. These instrumentations support specific libraries like the AWS SDK and HTTP Clients, as well as web frameworks like Spring Boot or Express.js.

OpenTelemetry's instrumentation libraries generate detailed spans for your libraries through library hooks or automatic code modification, requiring minimal code changes.

To determine if OpenTelemetry's Library Instrumentations supports your library, search for it in the OpenTelemetry Registry at OpenTelemetry Registry.

Exporting traces

X-Ray and OpenTelemetry use different methods to export trace data.

X-Ray trace export

The X-Ray SDKs use an emitter to send trace data:

- Sends segments and subsegments to the X-Ray Daemon
- Uses UDP for non-blocking I/O
- Configured by default in the SDK

OpenTelemetry trace export

OpenTelemetry uses configurable *Span Exporters* to send trace data:

- Uses *http/protobuf* or *grpc* protocols
- Exports spans to endpoints monitored by the OpenTelemetry Collector or CloudWatch Agent
- Allows for custom exporter configurations

Processing and forwarding traces

Both X-Ray and OpenTelemetry provide components to receive, process, and forward trace data.

X-Ray trace processing

The X-Ray Daemon handles trace processing:

- Listens for UDP traffic from X-Ray SDKs
- Batches segments and subsegments
- Uploads batches to the X-Ray service

OpenTelemetry trace processing

The OpenTelemetry Collector handles trace processing:

- Receives traces from instrumented services
- Processes and optionally modifies trace data
- Sends processed traces to various backends, including X-Ray

> **Note**
>
> The AWS CloudWatch Agent can also receive and send OpenTelemetry traces to X-Ray. For more information, see Collect metrics and traces with OpenTelemetry.

Span processing (OpenTelemetry-specific concept)

OpenTelemetry uses Span Processors to modify spans as they're created:

- Allows reading and modifying spans at creation or completion
- Enables custom logic for span handling

Baggage (OpenTelemetry-soecific concept)

OpenTelemetry's Baggage feature allows propagation of key-value data:

- Enables passing arbitrary data alongside trace context
- Useful for propagating application-specific information across service boundaries

For information about the OpenTelemetry Collector, see OpenTelemetry Collector

For information about X-Ray concepts, see X-Ray concepts in the X-Ray Developer Guide

Migration overview

This section provides an overview of the code changes required for migration. The list below are language-specific guidance and X-Ray Daemon migration steps.

⚠ **Important**

To fully migrate from X-Ray instrumentation to OpenTelemetry instrumentation, you need to:

1. Replace X-Ray SDK usage with an OpenTelemetry solution
2. Replace the X-Ray Daemon with the CloudWatch Agent or OpenTelemetry Collector (with X-Ray Exporter)

- Migrating to OpenTelemetry Java
- Migrate to OpenTelemetry Go
- Migrate to OpenTelemetry Node.js
- Migrate to OpenTelemetry .NET
- Migrate to OpenTelemetry Python
- Migrate to OpenTelemetry Ruby

Recommendations for new and existing applications

For new and existing applications, it is recommended to use the following solutions to enable tracing in your applications:

Instrumentation

- OpenTelemetry SDKs
- AWS Distro for OpenTelemetry Instrumentation

Data Collection

- OpenTelemetry Collector
- CloudWatch Agent

After migrating to OpenTelemetry-based solutions, your CloudWatch experience will remain the same. You will still be able to view your traces in the same format in the CloudWatch console's Traces and Trace Map pages, or retrieve your trace data through the X-Ray APIs.

Tracing setup changes

You need to replace the X-Ray setup with an OpenTelemetry setup.

Comparison of X-Ray and OpenTelemetry setup

Feature	X-Ray SDK	OpenTelemetry
Default configurations	• X-Ray Centralized Sampling • X-Ray Trace Context propagation • Trace Export to X-Ray Daemon	• Exporting traces to OpenTelemetry Collector or CloudWatch Agent (HTTP/gRPC) • W3C Trace Context propagation
Manual configurations	• Local sampling rules • Resource detection plug-ins	• X-Ray Sampling (may not be available for all languages) • Resource detection • X-Ray Trace Context propagation

Library instrumentation changes

Update your code to use OpenTelemetry Library Instrumentation instead of X-Ray Library Instrumentation for AWS SDK, HTTP Clients, Web Frameworks, and other libraries. This generates OpenTelemetry Traces instead of X-Ray Traces.

> **Note**
>
> Code changes vary by language and library. Refer to the language-specific migration guides for detailed instructions.

Lambda environment instrumentation changes

To use OpenTelemetry in your Lambda functions, choose one of these setup options:

1. Use an auto-instrumentation Lambda Layer:
   - (Recommended) CloudWatch Application Signals Lambda layer

     > **Note**
     >
     > To use only tracing, set the Lambda environment variable `OTEL_AWS_APPLICATION_SIGNALS_ENABLED=false`.

   - AWS managed Lambda Layer for ADOT
2. Manually set up OpenTelemetry for your Lambda function:
   - Configure a Simple Span Processor with an X-Ray UDP Span Exporter
   - Set up an X-Ray Lambda propagator

Manually creating trace data

Replace X-Ray segments and sub-segments with OpenTelemetry Spans:

- Use an OpenTelemetry Tracer to create Spans
- Add attributes to Spans (equivalent to X-Ray metadata and annotations)

> **Important**
>
> When sent to X-Ray:
>
> - Server Spans convert to X-Ray segments
> - Other Spans convert to X-Ray sub-segments
> - Attributes convert to metadata by default

To convert an attribute to an annotation, add its key to the `aws.xray.annotations` attribute list. For more information, see Enable Customized X-Ray Annotations.

Migrating from X-Ray Daemon to AWS CloudWatch agent or OpenTelemetry collector

You can use either the CloudWatch agent or OpenTelemetry collector to receive traces from your instrumented applications and send them to X-Ray.

Note

The CloudWatch agent version 1.300025.0 and later can collect OpenTelemetry traces. Using the CloudWatch agent instead of the X-Ray Daemon reduces the number of agents you need to manage. For more information, see Collecting metrics, logs, and traces with the CloudWatch agent.

Sections

- Migrating on Amazon EC2 or on-premises servers
- Migrating on Amazon ECS
- Migrating on Elastic Beanstalk

Migrating on Amazon EC2 or on-premises servers

Important

Stop the X-Ray Daemon process before using the CloudWatch agent or OpenTelemetry collector to prevent port conflicts.

Existing X-Ray Daemon setup

Installing the daemon

Your existing X-Ray Daemon usage was installed using one of these methods:

Manual installation

Download and run the executable file from the X-Ray daemon Amazon S3 bucket.

Automatic installation

Use this script to install the daemon when launching an instance:

```
#!/bin/bash
curl https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/xray-daemon/aws-
xray-daemon-3.x.rpm \
    -o /home/ec2-user/xray.rpm
yum install -y /home/ec2-user/xray.rpm
```

Configuring the daemon

Your existing X-Ray Daemon usage was configured using either:

- Command line arguments
- Configuration file (`xray-daemon.yaml`)

Example Using a configuration file

```
./xray -c ~/xray-daemon.yaml
```

Running the daemon

Your existing X-Ray Daemon usage was started with the following command:

```
~/xray-daemon$ ./xray -o -n us-east-1
```

Removing the daemon

To remove the X-Ray Daemon from your Amazon EC2 instance:

1. Stop the daemon service:

   ```
   systemctl stop xray
   ```

2. Delete the configuration file:

   ```
   rm ~/path/to/xray-daemon.yaml
   ```

3. If configured, remove the log file:

Note

The log file location depends on your configuration:

- Command line configuration: `/var/log/xray-daemon.log`
- Configuration file: Check the `LogPath` setting

Setting up the CloudWatch agent

Installing the agent

For installation instructions, see Installing the CloudWatch agent on an on-premises server.

Configuring the agent

1. Create a configuration file to enable trace collection. For more information, see Creating the CloudWatch agent configuration file.
2. Set up IAM permissions:
   - Attach an IAM role or specify credentials for the agent. For more information, see Setting up IAM roles.
   - Make sure the role or credentials include the `xray:PutTraceSegments` permission.

Starting the agent

For instructions to start the agent, see Starting the CloudWatch agent using the command line.

Setting up the OpenTelemetry collector

Installing the collector

Download and install the OpenTelemetry collector for your operating system. For instructions, see Installing the collector.

Configuring the collector

Configure the following components in your collector:

- awsproxy extension

  Required for X-Ray sampling

- OTel receivers

  Collects traces from your application
- xray exporter

  Sends traces to X-Ray

Example Sample collector configuration — otel-collector-config.yaml

```
extensions:
  awsproxy:
    endpoint: 127.0.0.1:2000
  health_check:

receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 127.0.0.1:4317
      http:
        endpoint: 127.0.0.1:4318

processors:
  batch:

exporters:
  awsxray:
    region: 'us-east-1'

service:
  pipelines:
    traces:
      receivers: [otlp]
      exporters: [awsxray]
  extensions: [awsproxy, health_check]
```

> ⚠ **Important**
>
> Configure AWS credentials with the `xray:PutTraceSegments` permission. For more information, see Specifying credentials.

Starting the collector

Run the collector with your configuration file:

```
otelcol --config=otel-collector-config.yaml
```

Migrating on Amazon ECS

> ⚠ **Important**
>
> Your task role must have the `xray:PutTraceSegments` permission for any collector you use.
> Stop any existing X-Ray Daemon container before running the CloudWatch agent or OpenTelemetry collector container on the same host to prevent port conflicts.

Using the CloudWatch agent

1. Get the Docker image from Amazon ECR Public Gallery.
2. Create a configuration file named `cw-agent-otel.json`:

```
{
  "traces": {
    "traces_collected": {
      "xray": {
        "tcp_proxy": {
          "bind_address": "0.0.0.0:2000"
        }
      },
      "otlp": {
        "grpc_endpoint": "0.0.0.0:4317",
        "http_endpoint": "0.0.0.0:4318"
      }
    }
  }
}
```

3. Store the configuration in Systems Manager Parameter Store:

   1. Open the https://console.aws.amazon.com/systems-manager/
   2. Choose **Create parameter**
   3. Enter the following values:
      - Name: `/ecs/cwagent/otel-config`
      - Tier: Standard
      - Type: String
      - Data type: Text
      - Value: [Paste the cw-agent-otel.json configuration here]

4. Create a task definition using bridge network mode:

   In your task definition, the configuration depends on the networking mode that you use. Bridge networking is the default and can be used in your default VPC. In a bridge network, set the `OTEL_EXPORTER_OTLP_TRACES_ENDPOINT`environment variable to tell the OpenTelemetry SDK what the endpoint and port are for the CloudWatch agent. You should also create a link from your application container to the Collector container for traces to be sent from the OpenTelemetry SDK in your application to the Collector container.

   Example CloudWatch agent task definition

```
{
    "containerDefinitions": [
        {
            "name": "cwagent",
            "image": "public.ecr.aws/cloudwatch-agent/cloudwatch-agent:latest",
            "portMappings": [
                {
                    "containerPort": 4318,
                    "hostPort": 4318,
                    "protocol": "tcp"
                },
                {
                    "containerPort": 4317,
                    "hostPort": 4317,
                    "protocol": "tcp"
                },
                {
                    "containerPort": 2000,
                    "hostPort": 2000,
                    "protocol": "tcp"
                }
            ],
            "secrets": [
                {
                    "name": "CW_CONFIG_CONTENT",
                    "valueFrom": "/ecs/cwagent/otel-config"
                }
            ]
        },
        {
            "name": "application",
            "image": "APPLICATION_IMAGE",
            "links": ["cwagent"],
            "environment": [
                {
                    "name": "OTEL_EXPORTER_OTLP_TRACES_ENDPOINT",
                    "value": "http://cwagent:4318/v1/traces"
                }
            ]
        }
    ]
}
```

For more information, see Deploying the CloudWatch agent to collect Amazon EC2 instance-level metrics on Amazon ECS .

Using the OpenTelemetry collector

1. Get the Docker image `otel/opentelemetry-collector-contrib` from Docker Hub.
2. Create a configuration file called `otel-collector-config.yaml` using the same content as shown in the **Amazon EC2 configuring the collector** section, but update the endpoints to use `0.0.0.0` instead of `127.0.0.1`.
3. To use this configuration in Amazon ECS, you can store the configuration in Systems Manager Parameter Store. First, go to Systems Manager Parameter Store console, and choose **Create new parameter** . Create a new parameter with the following information:

- Name: /ecs/otel/config (this name will be referenced in the Task Definition for the Collector)
- Tier: Standard
- Type: String
- Data type: Text
- Value: [Paste the otel-collector-config.yaml configuration here]

4. Create a task definition to deploy the OpenTelemetry collector using the bridge network mode as an example.

   In the task definition, the configuration depends on the networking mode that you use. Bridge networking is the default and can be used in your default VPC. In a bridge network, set the `OTEL_EXPORTER_OTLP_TRACES_ENDPOINT` environment variable to tell the OpenTelemetry SDK what the endpoint and port are for the OpenTelemetry Collector. You should also create a link from your application container to the Collector container for traces to be sent from the OpenTelemetry SDK in your application to the Collector container.

   Example OpenTelemetry collector task definition

```
{
    "containerDefinitions": [
        {
            "name": "otel-collector",
            "image": "otel/opentelemetry-collector-contrib",
            "portMappings": [
                {
                    "containerPort": 2000,
                    "hostPort": 2000
                },
                {
                    "containerPort": 4317,
                    "hostPort": 4317
                },
                {
                    "containerPort": 4318,
                    "hostPort": 4318
                }
            ],
            "command": [
                "--config",
                "env:SSM_CONFIG"
            ],
            "secrets": [
                {
                    "name": "SSM_CONFIG",
                    "valueFrom": "/ecs/otel/config"
                }
            ]
        },
        {
            "name": "application",
            "image": "APPLICATION_IMAGE",
            "links": ["otel-collector"],
            "environment": [
                {
                    "name": "OTEL_EXPORTER_OTLP_TRACES_ENDPOINT",
                    "value": "http://otel-collector:4318/v1/traces"
                }
            ]
        }
    ]
}
```

Migrating on Elastic Beanstalk

Important

Stop the X-Ray Daemon process before using the CloudWatch agent to prevent port conflicts.

Your existing X-Ray Daemon integration was turned on by using the Elastic Beanstalk console, or by configuring X-Ray Daemon in your application source code with a configuration file.

Using the CloudWatch agent

On the Amazon Linux 2 platform, configure the CloudWatch agent using an `.ebextensions` configuration file:

1. Create a directory named `.ebextensions` in your project root
2. Create a file named `cloudwatch.config` within the `.ebextensions` directory with the following content:

```
files:
  "/opt/aws/amazon-cloudwatch-agent/etc/config.json":
    mode: "0644"
    owner: root
    group: root
    content: |
      {
        "traces": {
          "traces_collected": {
            "otlp": {
              "grpc_endpoint": "12.0.0.1:4317",
              "http_endpoint": "12.0.0.1:4318"
            }
          }
        }
      }
container_commands:
  start_agent:
    command: /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl -a
 append-config -c file:/opt/aws/amazon-cloudwatch-agent/etc/config.json -s
```

3. Include the `.ebextensions` directory in your application source bundle when you deploy

For more information about Elastic Beanstalk configuration files, see Advanced environment customization with configuration files.

Migrating to OpenTelemetry Java

This section provides guidance on migrating from the X-Ray SDK to the OpenTelemetry SDK for Java applications.

Sections

- Zero code automatic instrumentation solution
- Manual instrumentation solutions with the SDK
- Tracing incoming requests (spring framework instrumentation)
- AWS SDK v2 instrumentation
- Instrumenting outgoing HTTP calls
- Instrumentation support for other libraries

- Manually creating trace data
- Lambda instrumentation

Zero code automatic instrumentation solution

With X-Ray Java agent

To enable the X-Ray Java agent, your application's JVM arguments were required to be modified.

```
-javaagent:/path-to-disco/disco-java-agent.jar=pluginPath=/path-to-disco/disco-
plugins
```

With OpenTelemetry-based Java agent

To use OpenTelemetry-based Java agents.

- Use the AWS Distro for OpenTelemetry (ADOT) Auto-Instrumentation Java agent for automatic instrumentation with the ADOT Java agent. For more information, see Auto-Instrumentation for Traces and Metrics with the Java agent. If you only want tracing, disable the `OTEL_METRICS_EXPORTER=none` environment variable. to export metrics from the Java agent.

  (Optional) You can also enable CloudWatch Application Signals when automatically instrumenting your applications on AWS with the ADOT Java auto-instrumentation to monitor current application health and track long-term application performance. Application Signals provides an unified, application-centric view of your applications, services, and dependencies, and helps monitor and triage application health. For more information, see Application Signals.
- Use the OpenTelemetry Java agent for automatic instrumentation. For more information, see Zero-code instrumentation with the Java Agent.

Manual instrumentation solutions with the SDK

Tracing setup with X-Ray SDK

To instrument your code with the X-Ray SDK for Java, first, the `AWSXRay` class was required to be configured with service plug-ins and local sampling rules, then a provided recorder was used.

```
static {
    AWSXRayRecorderBuilder builder =
 AWSXRayRecorderBuilder.standard().withPlugin(new EC2Plugin()).withPlugin(new
 ECSPlugin());
    AWSXRay.setGlobalRecorder(builder.build());
}
```

Tracing setup with OpenTelemetry SDK

The following dependencies are required.

```
<dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>io.opentelemetry</groupId>
                <artifactId>opentelemetry-bom</artifactId>
                <version>1.49.0</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <dependency>
                <groupId>io.opentelemetry.instrumentation</groupId>
                <artifactId>opentelemetry-instrumentation-bom</artifactId>
                <version>2.15.0</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-sdk</artifactId>
        </dependency>
        <dependency>
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-api</artifactId>
        </dependency>
        <dependency>
            <groupId>io.opentelemetry.semconv</groupId>
            <artifactId>opentelemetry-semconv</artifactId>
        </dependency>
        <dependency>
```

```
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-exporter-otlp</artifactId>
        </dependency>
        <dependency>
            <groupId>io.opentelemetry.contrib</groupId>
            <artifactId>opentelemetry-aws-xray</artifactId>
            <version>1.46.0</version>
        </dependency>
        <dependency>
            <groupId>io.opentelemetry.contrib</groupId>
            <artifactId>opentelemetry-aws-xray-propagator</artifactId>
            <version>1.46.0-alpha</version>
        </dependency>
        <dependency>
            <groupId>io.opentelemetry.contrib</groupId>
            <artifactId>opentelemetry-aws-resources</artifactId>
            <version>1.46.0-alpha</version>
        </dependency>
    </dependencies>
```

Configure the OpenTelemetry SDK by instantiating a `TracerProvider` and globally register an `OpenTelemetrySdk` object. Configure these components:

- An OTLP Span Exporter (for example, OtlpGrpcSpanExporter) - Required for exporting traces to the CloudWatch agent or OpenTelemetry Collector
- An AWS X-Ray Propagator – Required for propagating the Trace Context to AWS Services that are integrated with X-Ray
- An AWS X-Ray Remote Sampler – Required if you need to sample requests using X-Ray Sampling Rules
- Resource Detectors(for example, EcsResource or Ec2Resource) – Detect metadata of the host running your application

```
import io.opentelemetry.api.common.Attributes;
import io.opentelemetry.context.propagation.ContextPropagators;
import io.opentelemetry.contrib.aws.resource.Ec2Resource;
import io.opentelemetry.contrib.aws.resource.EcsResource;
import io.opentelemetry.contrib.awsxray.AwsXrayRemoteSampler;
import io.opentelemetry.contrib.awsxray.propagator.AwsXrayPropagator;
import io.opentelemetry.exporter.otlp.trace.OtlpGrpcSpanExporter;
import io.opentelemetry.sdk.OpenTelemetrySdk;
import io.opentelemetry.sdk.resources.Resource;
import io.opentelemetry.sdk.trace.SdkTracerProvider;
import io.opentelemetry.sdk.trace.export.BatchSpanProcessor;
import io.opentelemetry.sdk.trace.samplers.Sampler;
import static io.opentelemetry.semconv.ServiceAttributes.SERVICE_NAME;

// ...

    private static final Resource otelResource =
        Resource.create(Attributes.of(SERVICE_NAME, "YOUR_SERVICE_NAME"))
            .merge(EcsResource.get())
            .merge(Ec2Resource.get());
    private static final SdkTracerProvider sdkTracerProvider =
        SdkTracerProvider.builder()
            .addSpanProcessor(BatchSpanProcessor.create(
                OtlpGrpcSpanExporter.getDefault()
            ))
            .addResource(otelResource)
            .setSampler(Sampler.parentBased(
                AwsXrayRemoteSampler.newBuilder(otelResource).build()
            ))
            .build();
    // Globally registering a TracerProvider makes it available throughout the
 application to create as many Tracers as needed.
    private static final OpenTelemetrySdk openTelemetry =
        OpenTelemetrySdk.builder()
            .setTracerProvider(sdkTracerProvider)

 .setPropagators(ContextPropagators.create(AwsXrayPropagator.getInstance()))
            .buildAndRegisterGlobal();
```

Tracing incoming requests (spring framework instrumentation)

With X-Ray SDK

For information on how to use the X-Ray SDK with the spring framework to instrument your application, see AOP with Spring and the X-Ray SDK for Java. To enable AOP in Spring, complete these steps.

1. Configure Spring
2. Adding a tracing filter to your application
3. Annotate your code or implement an interface

4. Activate X-Ray in your application

With OpenTelemetry SDK

OpenTelemetry provides instrumentation libraries to collect traces for incoming requests for Spring Boot applications. To enable Spring Boot instrumentation with minimal configuration, include the following dependency.

```
<dependency>
           <groupId>io.opentelemetry.instrumentation</groupId>
            <artifactId>opentelemetry-spring-boot-starter</artifactId>
        </dependency>
```

For more information on how to enable and configure Spring Boot instrumentation for your OpenTelemetry setup, see OpenTelemetry's Getting started.

Using OpenTelemetry-based Java agents

The default recommended method for instrumenting Spring Boot applications is by using the OpenTelemetry Java agent with *bytecode* instrumentation, which also provides more out-of-the-box instrumentations and configurations when compared to directly using the SDK. For get started, see Zero code automatic instrumentation solution.

AWS SDK v2 instrumentation

With X-Ray SDK

The X-Ray SDK for Java can automatically instrument all AWS SDK v2 clients when you added the `aws-xray-recorder-sdk-aws-sdk-v2-instrumentor` sub-module in your build.

To instrument individual clients downstream client calls to AWS services with AWS SDK for Java 2.2 and later, the `aws-xray-recorder-sdk-aws-sdk-v2-instrumentor` module from your build configuration was excluded and the `aws-xray-recorder-sdk-aws-sdk-v2` module was included. Individual clients were instrumented by configuring them with a `TracingInterceptor`.

```
import com.amazonaws.xray.interceptors.TracingInterceptor;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
```

```
//...

public class MyModel {
  private DynamoDbClient client = DynamoDbClient.builder()
    .region(Region.US_WEST_2)
    .overrideConfiguration(
      ClientOverrideConfiguration.builder()
        .addExecutionInterceptor(new TracingInterceptor())
        .build()
      )
    .build();
//...
```

With OpenTelemetry SDK

To automatically instrument all AWS SDK clients, add the `opentelemetry-aws-sdk-2.2-autoconfigure` sub-module.

```
<dependency>
            <groupId>io.opentelemetry.instrumentation</groupId>
            <artifactId>opentelemetry-aws-sdk-2.2-autoconfigure</artifactId>
            <version>2.15.0-alpha</version>
            <scope>runtime</scope>
        </dependency>
```

To instrument individual AWSSDK clients, add the `opentelemetry-aws-sdk-2.2` sub-module.

```
<dependency>
            <groupId>io.opentelemetry.instrumentation</groupId>
            <artifactId>opentelemetry-aws-sdk-2.2</artifactId>
            <version>2.15.0-alpha</version>
            <scope>compile</scope>
        </dependency>
```

Then, register an interceptor when creating an AWS SDK Client.

```
import io.opentelemetry.instrumentation.awssdk.v2_2.AwsSdkTelemetry;

// ...
```

```
    AwsSdkTelemetry telemetry = AwsSdkTelemetry.create(openTelemetry);
    private final S3Client S3_CLIENT = S3Client.builder()
      .overrideConfiguration(ClientOverrideConfiguration.builder()
        .addExecutionInterceptor(telemetry.newExecutionInterceptor())
        .build())
      .build();
```

Instrumenting outgoing HTTP calls

With X-Ray SDK

To instrument outgoing HTTP requests with X-Ray, the X-Ray SDK for Java's version of the Apache HttpClient was required.

```
import com.amazonaws.xray.proxies.apache.http.HttpClientBuilder;
...
  public String randomName() throws IOException {
    CloseableHttpClient httpclient = HttpClientBuilder.create().build();
```

With OpenTelemetry SDK

Similarly to the X-Ray Java SDK, OpenTelemetry provides an `ApacheHttpClientTelemetry` class that has a builder method that allows creation of an instance of an the `HttpClientBuilder` to provide OpenTelemetry-based spans and context propagation for Apache HttpClient.

```
<dependency>
            <groupId>io.opentelemetry.instrumentation</groupId>
            <artifactId>opentelemetry-apache-httpclient-5.2</artifactId>
            <version>2.15.0-alpha</version>
            <scope>compile</scope>
        </dependency>
```

The following is a code example from the opentelemetry-java-instrumentation . The HTTP Client provided by newHttpClient() will generate traces for executed requests.

```
import io.opentelemetry.api.OpenTelemetry;
import
 io.opentelemetry.instrumentation.apachehttpclient.v5_2.ApacheHttpClientTelemetry;
```

```
import org.apache.hc.client5.http.classic.HttpClient;
import org.apache.hc.client5.http.impl.classic.HttpClientBuilder;

public class ApacheHttpClientConfiguration {

  private OpenTelemetry openTelemetry;

  public ApacheHttpClientConfiguration(OpenTelemetry openTelemetry) {
    this.openTelemetry = openTelemetry;
  }

  // creates a new http client builder for constructing http clients with open
 telemetry instrumentation
  public HttpClientBuilder createBuilder() {
    return
 ApacheHttpClientTelemetry.builder(openTelemetry).build().newHttpClientBuilder();
  }

  // creates a new http client with open telemetry instrumentation
  public HttpClient newHttpClient() {
    return ApacheHttpClientTelemetry.builder(openTelemetry).build().newHttpClient();
  }
}
```

Instrumentation support for other libraries

Find the full list of supported Library instrumentations for OpenTelemetry Java in its respective instrumentation GitHub repository , under Supported libraries, frameworks, application servers, and JVMs .

Alternatively, you can search the OpenTelemetry Registry to find out if OpenTelemetry supports instrumentation. To start searching, see Registry.

Manually creating trace data

With X-Ray SDK

With the X-Ray SDK, the `beginSegment` and `beginSubsegment` methods are needed to manually create X-Ray segments and sub-segments.

```
  Segment segment = xrayRecorder.beginSegment("ManualSegment");
```

```
        segment.putAnnotation("annotationKey", "annotationValue");
        segment.putMetadata("metadataKey", "metadataValue");

        try {
            Subsegment subsegment =
 xrayRecorder.beginSubsegment("ManualSubsegment");
            subsegment.putAnnotation("key", "value");

            // Do something here

        } catch (Exception e) {
            subsegment.addException(e);
        } finally {
            xrayRecorder.endSegment();
        }
```

With OpenTelemetry SDK

You can use custom spans to monitor the performance of internal activities that are not captured by instrumentation libraries. Note that only span kind server are converted into X-Ray segments, all other spans are converted into X-Ray sub-segments.

First, you will need to create a *Tracer* in order to generate spans, which you can obtain through the `openTelemetry.getTracer` method. This will provide a Tracer instance from the `TracerProvider` that was registered globally in the Manual instrumentation solutions with the SDK example. You can create as many Tracer instances as needed, but it is common to have one Tracer for an entire application.

```
Tracer tracer = openTelemetry.getTracer("my-app");
```

You can use the Tracer to create spans.

```
import io.opentelemetry.api.common.AttributeKey;
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.api.trace.SpanKind;
import io.opentelemetry.api.trace.Tracer;
import io.opentelemetry.context.Scope;

...

// SERVER span will become an X-Ray segment
```

```
Span span = tracer.spanBuilder("get-token")
  .setKind(SpanKind.SERVER)
  .setAttribute("key", "value")
  .startSpan();
try (Scope ignored = span.makeCurrent()) {

  span.setAttribute("metadataKey", "metadataValue");
  span.setAttribute("annotationKey", "annotationValue");

  // The following ensures that "annotationKey: annotationValue" is an annotation in
 X-Ray raw data.
  span.setAttribute(AttributeKey.stringArrayKey("aws.xray.annotations"),
 List.of("annotationKey"));

  // Do something here
}

span.end();
```

Spans have a default type of *INTERNAL*.

```
// Default span of type INTERNAL will become an X-Ray subsegment
Span span = tracer.spanBuilder("process-header")
  .startSpan();
try (Scope ignored = span.makeCurrent()) {
  doProcessHeader();
}
```

Adding annotations and metadata to traces with OpenTelemetry SDK

In the above example, the `setAttribute` method is used to add attributes to each span. By default, all the span attributes will be converted into metadata in X-Ray raw data. To ensure that an attribute is converted into an annotation and not metadata, the above example adds that attribute's key to the list of the `aws.xray.annotations` attribute. For more information, see Enable the Customized X-Ray Annotations and Annotations and metadata.

With OpenTelemetry-based Java agents

If you are using the Java agent to automatically instrument your application, you need to perform manual instrumentation in your application. For example, to instrument code within the application for sections that are not covered by any auto-instrumentation library.

To perform manual instrumentation with the agent, you need to use the `opentelemetry-api` artifact. The artifact version cannot be newer than the agent version.

```
import io.opentelemetry.api.GlobalOpenTelemetry;
import io.opentelemetry.api.trace.Span;

// ...

        Span parentSpan = Span.current();
        Tracer tracer = GlobalOpenTelemetry.getTracer("my-app");
        Span span = tracer.spanBuilder("my-span-name")
            .setParent(io.opentelemetry.context.Context.current().with(parentSpan))
            .startSpan();
        span.end();
```

Lambda instrumentation

With X-Ray SDK

Using the X-Ray SDK, after your Lambda has *Active Tracing* enabled, there is no additional configuration required to use the X-Ray SDK. Lambda will create a segment representing the Lambda handler invocation, and you can create sub-segments or instrument libraries using the X-Ray SDK without any additional configuration.

With OpenTelemetry-based solutions

Auto-instrumentation Lambda layers – You can automatically instrument your Lambda with AWS vended Lambda layers using the following solutions:

- CloudWatch Application Signals Lambda layer (Recommended)

> **Note**
>
> This Lambda layer has CloudWatch Application Signals enabled by default, which enables performance and health monitoring for your Lambda application by collecting both metrics and traces. For just tracing, set the Lambda environment variable `OTEL_AWS_APPLICATION_SIGNALS_ENABLED=false`.

  - Enables performance and health monitoring for your Lambda application

    - Collects both metrics and traces by default
- AWS managed Lambda layer for ADOT Java. For more information, see AWS Distro for OpenTelemetry Lambda Support For Java.

To use manual instrumentation along with auto-instrumentation layer, see Manual instrumentation solutions with the SDK. For reduced cold starts, consider using OpenTelemetry manual instrumentation to generate OpenTelemetry traces for your Lambda function.

OpenTelemetry manual instrumentation for AWS Lambda

Consider the following Lambda function code that makes an Amazon S3 ListBuckets call.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.ListBucketsRequest;
import software.amazon.awssdk.services.s3.model.ListBucketsResponse;
import software.amazon.awssdk.services.s3.model.S3Exception;

public class ListBucketsLambda implements RequestHandler<String, String> {

    private final S3Client S3_CLIENT = S3Client.builder()
        .build();

    @Override
    public String handleRequest(String input, Context context) {
        try {
            ListBucketsResponse response = makeListBucketsCall();
            context.getLogger().log("response: " + response.toString());
            return "Success";
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private ListBucketsResponse makeListBucketsCall() {
        try {
            ListBucketsRequest listBucketsRequest = ListBucketsRequest.builder()
```

```
                .build();
            ListBucketsResponse response = S3_CLIENT.listBuckets(listBucketsRequest);
            return response;
        } catch (S3Exception e) {
            throw new RuntimeException("Failed to call S3 listBuckets" +
 e.awsErrorDetails().errorMessage(), e);
        }
    }
}
```

Here are the dependencies.

```
dependencies {
    implementation('com.amazonaws:aws-lambda-java-core:1.2.3')
    implementation('software.amazon.awssdk:s3:2.28.29')
    implementation('org.slf4j:slf4j-nop:2.0.16')
}
```

To manually instrument your Lambda handler and the Amazon S3 client, do the following.

1. Replace your function classes that implement `RequestHandler` (or RequestStreamHandler) with those that extend `TracingRequestHandler` (or TracingRequestStreamHandler).
2. Instantiate a TracerProvider and globally register an OpenTelemetrySdk object. The TracerProvider is recommended to be configured with:

    a. A Simple Span Processor with an X-Ray UDP span exporter to send Traces to Lambda’s UDP X-Ray endpoint

    b. A ParentBased always on sampler (Default if not configured)

    c. A Resource with service.name set to the Lambda function name

    d. An X-Ray Lambda propagator
3. Change the `handleRequest` method to `doHandleRequest` and pass the `OpenTelemetrySdk` object to the base class.
4. Instrument the Amazon S3 client with the OpenTemetry AWS SDK instrumentation by registering the interceptor when building the client.

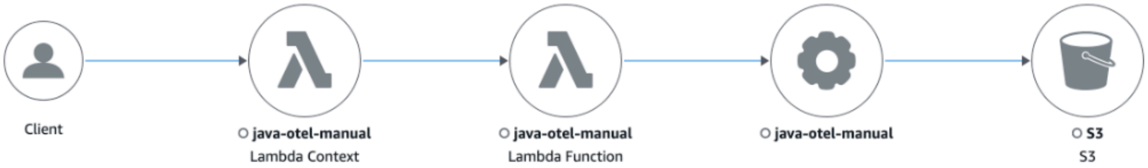
You need the following OpenTelemetry-related dependencies.

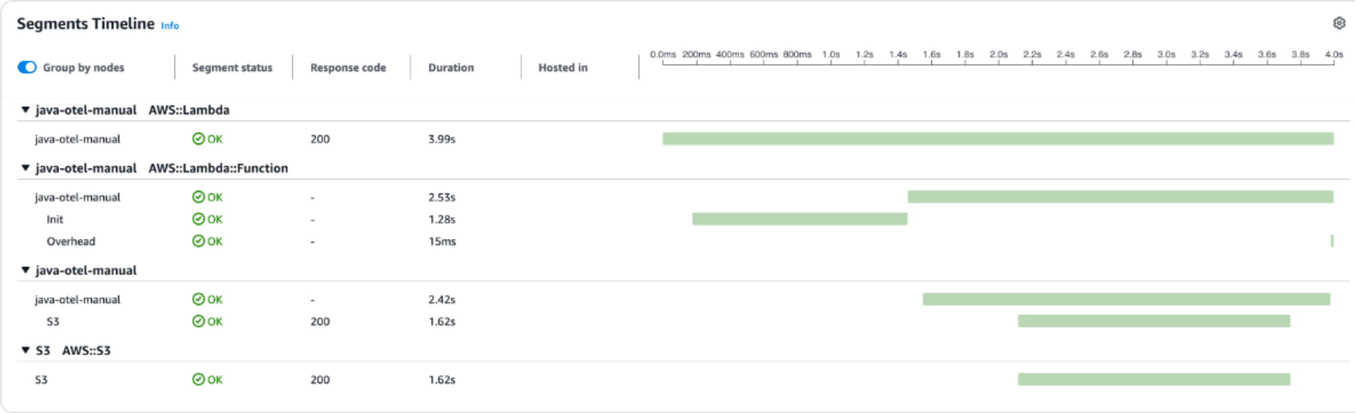
```
dependencies {
    ...

    implementation("software.amazon.distro.opentelemetry:aws-distro-opentelemetry-xray-
udp-span-exporter:0.1.0")

    implementation(platform('io.opentelemetry.instrumentation:opentelemetry-
instrumentation-bom:2.14.0'))
    implementation(platform('io.opentelemetry:opentelemetry-bom:1.48.0'))

    implementation('io.opentelemetry:opentelemetry-sdk')
    implementation('io.opentelemetry:opentelemetry-api')
    implementation('io.opentelemetry.contrib:opentelemetry-aws-xray-propagator:1.45.0-
alpha')
    implementation('io.opentelemetry.contrib:opentelemetry-aws-resources:1.45.0-alpha')
    implementation('io.opentelemetry.instrumentation:opentelemetry-aws-lambda-
core-1.0:2.14.0-alpha')
    implementation('io.opentelemetry.instrumentation:opentelemetry-aws-sdk-2.2:2.14.0-
alpha')
}
```

The following code demonstrates the Lambda function after the required changes. You can create additional custom spans to complement the automatically provided spans.

```
package example;

import java.time.Duration;

import com.amazonaws.services.lambda.runtime.Context;

import io.opentelemetry.api.common.Attributes;
import io.opentelemetry.context.propagation.ContextPropagators;
import io.opentelemetry.contrib.aws.resource.LambdaResource;
import io.opentelemetry.contrib.awsxray.propagator.AwsXrayLambdaPropagator;
import io.opentelemetry.instrumentation.awslambdacore.v1_0.TracingRequestHandler;
import io.opentelemetry.instrumentation.awssdk.v2_2.AwsSdkTelemetry;
import io.opentelemetry.sdk.OpenTelemetrySdk;
import io.opentelemetry.sdk.resources.Resource;
import io.opentelemetry.sdk.trace.SdkTracerProvider;
import io.opentelemetry.sdk.trace.export.SimpleSpanProcessor;
import io.opentelemetry.sdk.trace.samplers.Sampler;
import static io.opentelemetry.semconv.ServiceAttributes.SERVICE_NAME;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.ListBucketsRequest;
import software.amazon.awssdk.services.s3.model.ListBucketsResponse;
import software.amazon.awssdk.services.s3.model.S3Exception;
import
 software.amazon.distro.opentelemetry.exporter.xray.udp.trace.AwsXrayUdpSpanExporterBuilder;

public class ListBucketsLambda extends TracingRequestHandler<String, String> {
    private static final Resource lambdaResource = LambdaResource.get();
    private static final SdkTracerProvider sdkTracerProvider =
        SdkTracerProvider.builder()
            .addSpanProcessor(SimpleSpanProcessor.create(
                new AwsXrayUdpSpanExporterBuilder().build()
            ))
            .addResource(
                lambdaResource
                .merge(Resource.create(Attributes.of(SERVICE_NAME,
 System.getenv("AWS_LAMBDA_FUNCTION_NAME"))))
            )
            .setSampler(Sampler.parentBased(Sampler.alwaysOn()))
            .build();
    private static final OpenTelemetrySdk openTelemetry =
        OpenTelemetrySdk.builder()
            .setTracerProvider(sdkTracerProvider)

 .setPropagators(ContextPropagators.create(AwsXrayLambdaPropagator.getInstance()))
            .buildAndRegisterGlobal();
    private static final AwsSdkTelemetry telemetry =
 AwsSdkTelemetry.create(openTelemetry);
    private final S3Client S3_CLIENT = S3Client.builder()
        .overrideConfiguration(ClientOverrideConfiguration.builder()
            .addExecutionInterceptor(telemetry.newExecutionInterceptor())
            .build())
        .build();

    public ListBucketsLambda() {
        super(openTelemetry, Duration.ofMillis(0));
    }

    @Override
    public String doHandleRequest(String input, Context context) {
        try {
            ListBucketsResponse response = makeListBucketsCall();
            context.getLogger().log("response: " + response.toString());
            return "Success";
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private ListBucketsResponse makeListBucketsCall() {
        try {
            ListBucketsRequest listBucketsRequest = ListBucketsRequest.builder()
                .build();
            ListBucketsResponse response = S3_CLIENT.listBuckets(listBucketsRequest);
            return response;
        } catch (S3Exception e) {
            throw new RuntimeException("Failed to call S3 listBuckets" +
 e.awsErrorDetails().errorMessage(), e);
        }
    }
}
```

When invoking the Lambda function, you will see the following trace under *Trace Map* in the CloudWatch console.




0.0ms 200ms 400ms 600ms 800ms 1.0s 1.2s 1.4s 1.6s 1.8s 2.0s 2.2s 2.4s 2.6s 2.8s 3.0s 3.2s 3.4s 3.6s 3.8s 4.0s

Segments Timeline Info

Group by nodes	Segment status	Response code	Duration	Hosted in
▼ java-otel-manual AWS::Lambda				
java-otel-manual	✓ OK	200	3.99s	
▼ java-otel-manual AWS::Lambda::Function				
java-otel-manual	✓ OK	-	2.53s	
Init	✓ OK	-	1.28s	
Overhead	✓ OK	-	15ms	
▼ java-otel-manual				
java-otel-manual	✓ OK	-	2.42s	
S3	✓ OK	200	1.62s	
▼ S3 AWS::S3				
S3	✓ OK	200	1.62s	

Migrate to OpenTelemetry Go

Use the following code examples to manually instrument your Go applications with the OpenTelemetry SDK when migrating from X-Ray.

Manual instrumentation with the SDK

Tracing setup with X-Ray SDK

When using the X-Ray SDK for Go, service plugins or local sampling rules were required to be configured before instrumenting your code.

```
func init() {
    if os.Getenv("ENVIRONMENT") == "production" {
        ec2.Init()
    }

    xray.Configure(xray.Config{
        DaemonAddr:       "127.0.0.1:2000",
        ServiceVersion:   "1.2.3",
    })
}
```

Set up tracing with OpenTelemetry SDK

Configure the OpenTelemetry SDK by instantiating a TracerProvider and registering it as the global tracer provider. We recommend configuring the following components:

- OTLP Trace Exporter – Required for exporting traces to the CloudWatch Agent or OpenTelemetry Collector
- X-Ray Propagator – Required for propagating the trace context to AWS services integrated with X-Ray
- X-Ray Remote Sampler – Required for sampling requests using X-Ray sampling rules
- Resource detectors – To detect metadata of the host running your application

```
import (
    "go.opentelemetry.io/contrib/detectors/aws/ec2"
    "go.opentelemetry.io/contrib/propagators/aws/xray"
    "go.opentelemetry.io/contrib/samplers/aws/xray"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracegrpc"
    "go.opentelemetry.io/otel/sdk/trace"
)

func setupTracing() error {
    ctx := context.Background()

    exporterEndpoint := os.Getenv("OTEL_EXPORTER_OTLP_ENDPOINT")
    if exporterEndpoint == "" {
        exporterEndpoint = "localhost:4317"
    }

    traceExporter, err := otlptracegrpc.New(ctx,
        otlptracegrpc.WithInsecure(),
        otlptracegrpc.WithEndpoint(exporterEndpoint))
    if err != nil {
        return fmt.Errorf("failed to create OTLP trace exporter: %v", err)
    }

    remoteSampler, err := xray.NewRemoteSampler(ctx, "my-service-name", "ec2")
    if err != nil {
        return fmt.Errorf("failed to create X-Ray Remote Sampler: %v", err)
    }

    ec2Resource, err := ec2.NewResourceDetector().Detect(ctx)
    if err != nil {
        return fmt.Errorf("failed to detect EC2 resource: %v", err)
    }

    tp := trace.NewTracerProvider(
        trace.WithSampler(remoteSampler),
        trace.WithBatcher(traceExporter),
        trace.WithResource(ec2Resource),
    )

    otel.SetTracerProvider(tp)
    otel.SetTextMapPropagator(xray.Propagator{})

    return nil
}
```

Tracing incoming requests (HTTP handler instrumentation)

With X-Ray SDK

To instrument an HTTP handler with X-Ray, the X-Ray handler method was used to generate segments using NewFixedSegmentNamer.

```
func main() {
    http.Handle("/", xray.Handler(xray.NewFixedSegmentNamer("myApp"),
 http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello!"))
    })))
    http.ListenAndServe(":8000", nil)
}
```

With OpenTelemetry SDK

To instrument an HTTP handler with OpenTelemetry, use the OpenTelemetry's newHandler method to wrap your original handler code.

```
import (
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"
)

helloHandler := func(w http.ResponseWriter, req *http.Request) {
    ctx := req.Context()
    span := trace.SpanFromContext(ctx)
    span.SetAttributes(attribute.Bool("isHelloHandlerSpan", true),
 attribute.String("attrKey", "attrValue"))

    _, _ = io.WriteString(w, "Hello World!\n")
}

otelHandler := otelhttp.NewHandler(http.HandlerFunc(helloHandler), "Hello")

http.Handle("/hello", otelHandler)
```

```
err = http.ListenAndServe(":8080", nil)
if err != nil {
    log.Fatal(err)
}
```

AWS SDK for Go v2 instrumentation

With X-Ray SDK

To instrument outgoing AWS requests from AWS SDK, your clients were instrumented as follows:

```

// Create a segment
ctx, root := xray.BeginSegment(context.TODO(), "AWSSDKV2_Dynamodb")
defer root.Close(nil)

cfg, err := config.LoadDefaultConfig(ctx, config.WithRegion("us-west-2"))
if err != nil {
    log.Fatalf("unable to load SDK config, %v", err)
}
// Instrumenting AWS SDK v2
awsv2.AWSV2Instrumentor(&cfg.APIOptions)
// Using the Config value, create the DynamoDB client
svc := dynamodb.NewFromConfig(cfg)
// Build the request with its input parameters
_, err = svc.ListTables(ctx, &dynamodb.ListTablesInput{
    Limit: aws.Int32(5),
})
if err != nil {
    log.Fatalf("failed to list tables, %v", err)
}
```

With OpenTelemetry SDK

Tracing support for downstream AWS SDK calls is provided by OpenTelemetry's AWS SDK for Go v2 Instrumentation. Here's an example of tracing an S3 client call:

```
import (
    ...

    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/s3"

    "go.opentelemetry.io/otel"
    oteltrace "go.opentelemetry.io/otel/trace"
    awsConfig "github.com/aws/aws-sdk-go-v2/config"
    "go.opentelemetry.io/contrib/instrumentation/github.com/aws/aws-sdk-go-v2/
otelaws"
)

...


    // init aws config
    cfg, err := awsConfig.LoadDefaultConfig(ctx)
    if err != nil {
        panic("configuration error, " + err.Error())
    }

    // instrument all aws clients
    otelaws.AppendMiddlewares(&.APIOptions)


    // Call to S3
    s3Client := s3.NewFromConfig(cfg)
    input := &s3.ListBucketsInput{}
    result, err := s3Client.ListBuckets(ctx, input)
    if err != nil {
        fmt.Printf("Got an error retrieving buckets, %v", err)
        return
    }
```

Instrumenting outgoing HTTP calls

With X-Ray SDK

To instrument outgoing HTTP calls with X-Ray, the xray.Client was used to create a copy of a provided HTTP client.

```
myClient := xray.Client(http-client)

resp, err := ctxhttp.Get(ctx, xray.Client(nil), url)
```

With OpenTelemetry SDK

To instrument HTTP clients with OpenTelemetry, use OpenTelemetry's otelhttp.NewTransport method to wrap the http.DefaultTransport.

```
import (
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"
)

// Create an instrumented HTTP client.
httpClient := &http.Client{
    Transport: otelhttp.NewTransport(
        http.DefaultTransport,
    ),
}

req, err := http.NewRequestWithContext(ctx, http.MethodGet, "https://api.github.com/
repos/aws-observability/aws-otel-go/releases/latest", nil)
if err != nil {
    fmt.Printf("failed to create http request, %v\n", err)
}
res, err := httpClient.Do(req)
if err != nil {
    fmt.Printf("failed to make http request, %v\n", err)
}
// Request body must be closed
defer func(Body io.ReadCloser) {
    err := Body.Close()
    if err != nil {
        fmt.Printf("failed to close http response body, %v\n", err)
    }
}(res.Body)
```

Instrumentation support for other libraries

You can find the full list of supported library instrumentations for OpenTelemetry Go under Instrumentation packages .

Alternatively, you can search the OpenTelemetry registry to find out if OpenTelemetry supports instrumentation for your library under Registry.

Manually creating trace data

With X-Ray SDK

With the X-Ray SDK, the BeginSegment and BeginSubsegment methods was required to manually create X-Ray segments and sub-segments.

```
// Start a segment
ctx, seg := xray.BeginSegment(context.Background(), "service-name")
// Start a subsegment
subCtx, subSeg := xray.BeginSubsegment(ctx, "subsegment-name")

// Add metadata or annotation here if necessary
xray.AddAnnotation(subCtx, "annotationKey", "annotationValue")
xray.AddMetadata(subCtx, "metadataKey", "metadataValue")

subSeg.Close(nil)
// Close the segment
seg.Close(nil)
```

With OpenTelemetry SDK

Use custom spans to monitor the performance of internal activities that are not captured by instrumentation libraries. Note that only spans of kind Server are converted into X-Ray segments, all other spans are converted into X-Ray sub-segments.

First, you will need to create a Tracer to generate spans, which you can obtain through the `otel.Tracer` method. This will provide a Tracer instance from the TracerProvider that was registered globally in the Tracing Setup example. You can create as many Tracer instances as needed, but it is common to have one Tracer for an entire application.

```
    tracer := otel.Tracer("application-tracer")
```

```
import (
    ...
    oteltrace "go.opentelemetry.io/otel/trace"
)

...

    var attributes = []attribute.KeyValue{
        attribute.KeyValue{Key: "metadataKey", Value:
 attribute.StringValue("metadataValue")},
        attribute.KeyValue{Key: "annotationKey", Value:
 attribute.StringValue("annotationValue")},
        attribute.KeyValue{Key: "aws.xray.annotations", Value:
 attribute.StringSliceValue([]string{"annotationKey"})},
    }

    ctx := context.Background()

    parentSpanContext, parentSpan := tracer.Start(ctx,
 "ParentSpan", oteltrace.WithSpanKind(oteltrace.SpanKindServer),
 oteltrace.WithAttributes(attributes...))
    _, childSpan := tracer.Start(parentSpanContext, "ChildSpan",
 oteltrace.WithSpanKind(oteltrace.SpanKindInternal))

    // ...

    childSpan.End()
    parentSpan.End()
```

Adding annotations and metadata to traces with OpenTelemetry SDK

In the above example, the `WithAttributes` method is used to add attributes to each span. Note that by default, all the span attributes are converted into metadata in X-Ray raw data. To ensure that an attribute is converted into an annotation and not metadata, add the attribute's key to the list of the `aws.xray.annotations` attribute. For more information, see Enable The Customized X-Ray Annotations .

Lambda manual instrumentation

With X-Ray SDK

With the X-Ray SDK, after your Lambda has *Active Tracing* was enabled, there were no additional configurations required to use the X-Ray SDK. Lambda created a segment representing the Lambda handler invocation, and you created sub-segments using the X-Ray SDK without any additional configuration.

With OpenTelemetry SDK

The following Lambda function code (without instrumentation) makes an Amazon S3 ListBuckets call and outgoing HTTP request.

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "io"
    "net/http"
    "os"

    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
    awsconfig "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/s3"
)

func lambdaHandler(ctx context.Context) (interface{}, error) {
    // Initialize AWS config.
    cfg, err := awsconfig.LoadDefaultConfig(ctx)
    if err != nil {
        panic("configuration error, " + err.Error())
    }

    s3Client := s3.NewFromConfig(cfg)

    // Create an HTTP client.
    httpClient := &http.Client{
        Transport: http.DefaultTransport,
    }

    input := &s3.ListBucketsInput{}
    result, err := s3Client.ListBuckets(ctx, input)
    if err != nil {
        fmt.Printf("Got an error retrieving buckets, %v", err)
    }
    fmt.Println("Buckets:")
    for _, bucket := range result.Buckets {
        fmt.Println(*bucket.Name + ": " + bucket.CreationDate.Format("2006-01-02
 15:04:05 Monday"))
    }
    fmt.Println("End Buckets.")

    req, err := http.NewRequestWithContext(ctx, http.MethodGet, "https://
api.github.com/repos/aws-observability/aws-otel-go/releases/latest", nil)
    if err != nil {
        fmt.Printf("failed to create http request, %v\n", err)
    }
    res, err := httpClient.Do(req)
    if err != nil {
        fmt.Printf("failed to make http request, %v\n", err)
    }
    defer func(Body io.ReadCloser) {
        err := Body.Close()
        if err != nil {
            fmt.Printf("failed to close http response body, %v\n", err)
        }
    }(res.Body)

    var data map[string]interface{}
    err = json.NewDecoder(res.Body).Decode(&data)
    if err != nil {
        fmt.Printf("failed to read http response body, %v\n", err)
    }
    fmt.Printf("Latest ADOT Go Release is '%s'\n", data["name"])

    return events.APIGatewayProxyResponse{
        StatusCode: http.StatusOK,
        Body:       os.Getenv("_X_AMZN_TRACE_ID"),
    }, nil
}

func main() {
    lambda.Start(lambdaHandler)
}
```

To manually instrument your Lambda handler and the Amazon S3 client, do the following:

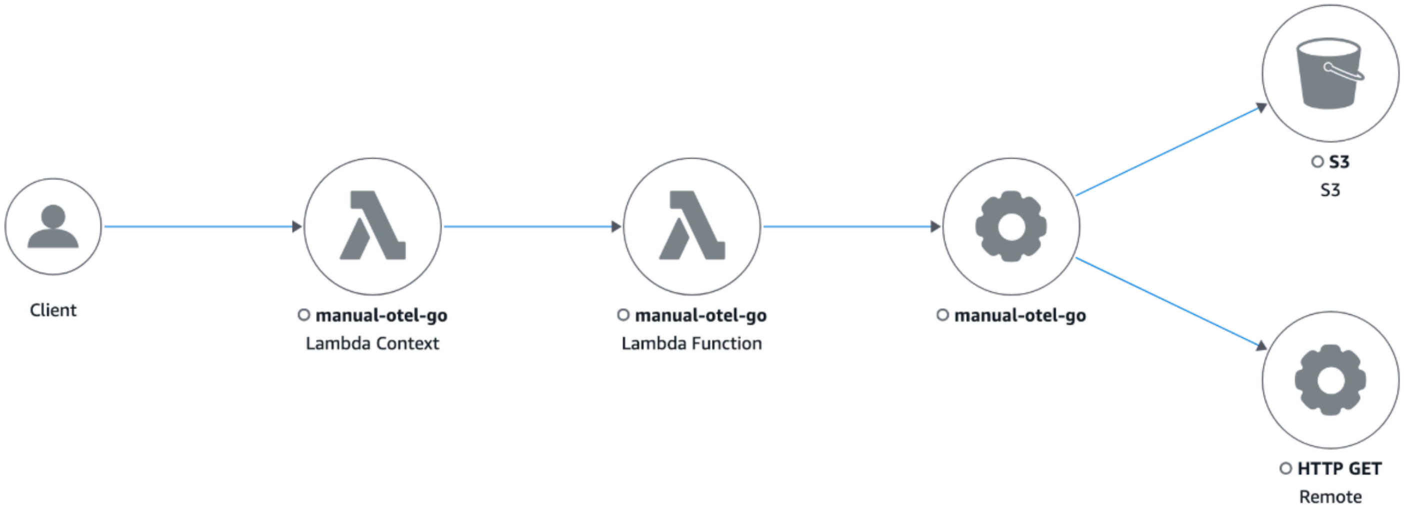
1. In *main()*, instantiate a TracerProvider (tp) and register it as the global tracer provider. The TracerProvider is recommended to be configured with:
    a. Simple Span Processor with an X-Ray UDP span exporter to send Traces to Lambda's UDP X-Ray endpoint
    b. Resource with *service.name* set to the Lambda function name
2. Change the usage of `lambda.Start(lambdaHandler)` to `lambda.Start(otellambda.InstrumentHandler(lambdaHandler, xrayconfig.WithRecommendedOptions(tp)...))`.
3. Instrument the Amazon S3 client with the OpenTemetry AWS SDK instrumentation by appending OpenTelemetry middleware for `aws-sdk-go-v2` into the Amazon S3 client configuration.
4. Instrument the http client by using OpenTelemetry's `otelhttp.NewTransport` method to wrap the `http.DefaultTransport`.

The following code is an example of how the Lambda Function will look like after the changes. You may manually create additional custom spans in addition to the spans provided automatically.

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "io"
    "net/http"
    "os"

    "github.com/aws-observability/aws-otel-go/exporters/xrayudp"
    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
    awsconfig "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/s3"

    lambdadetector "go.opentelemetry.io/contrib/detectors/aws/lambda"
    "go.opentelemetry.io/contrib/instrumentation/github.com/aws/aws-lambda-go/
otellambda"
    "go.opentelemetry.io/contrib/instrumentation/github.com/aws/aws-lambda-go/
otellambda/xrayconfig"
```

```
    "go.opentelemetry.io/contrib/instrumentation/github.com/aws/aws-sdk-go-v2/
otelaws"
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"
    "go.opentelemetry.io/contrib/propagators/aws/xray"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/sdk/resource"
    "go.opentelemetry.io/otel/sdk/trace"
    semconv "go.opentelemetry.io/otel/semconv/v1.26.0"
)

func lambdaHandler(ctx context.Context) (interface{}, error) {
    // Initialize AWS config.
    cfg, err := awsconfig.LoadDefaultConfig(ctx)
    if err != nil {
        panic("configuration error, " + err.Error())
    }

    // Instrument all AWS clients.
    otelaws.AppendMiddlewares(&cfg.APIOptions)
    // Create an instrumented S3 client from the config.
    s3Client := s3.NewFromConfig(cfg)

    // Create an instrumented HTTP client.
    httpClient := &http.Client{
        Transport: otelhttp.NewTransport(
            http.DefaultTransport,
        ),
    }

    // return func(ctx context.Context) (interface{}, error) {
    input := &s3.ListBucketsInput{}
    result, err := s3Client.ListBuckets(ctx, input)
    if err != nil {
        fmt.Printf("Got an error retrieving buckets, %v", err)
    }

    fmt.Println("Buckets:")
    for _, bucket := range result.Buckets {
        fmt.Println(*bucket.Name + ": " + bucket.CreationDate.Format("2006-01-02
 15:04:05 Monday"))
    }
    fmt.Println("End Buckets.")

    req, err := http.NewRequestWithContext(ctx, http.MethodGet, "https://
api.github.com/repos/aws-observability/aws-otel-go/releases/latest", nil)
    if err != nil {
        fmt.Printf("failed to create http request, %v\n", err)
    }
    res, err := httpClient.Do(req)
    if err != nil {
        fmt.Printf("failed to make http request, %v\n", err)
    }
    defer func(Body io.ReadCloser) {
        err := Body.Close()
        if err != nil {
            fmt.Printf("failed to close http response body, %v\n", err)
        }
    }(res.Body)

    var data map[string]interface{}
    err = json.NewDecoder(res.Body).Decode(&data)
    if err != nil {
        fmt.Printf("failed to read http response body, %v\n", err)
    }
    fmt.Printf("Latest ADOT Go Release is '%s'\n", data["name"])

    return events.APIGatewayProxyResponse{
        StatusCode: http.StatusOK,
        Body:       os.Getenv("_X_AMZN_TRACE_ID"),
    }, nil
}

func main() {
    ctx := context.Background()
    detector := lambdadetector.NewResourceDetector()
    lambdaResource, err := detector.Detect(context.Background())
    if err != nil {
        fmt.Printf("failed to detect lambda resources: %v\n", err)
    }

    var attributes = []attribute.KeyValue{
        attribute.KeyValue{Key: semconv.ServiceNameKey, Value:
 attribute.StringValue(os.Getenv("AWS_LAMBDA_FUNCTION_NAME"))},
    }
    customResource := resource.NewWithAttributes(semconv.SchemaURL, attributes...)
    mergedResource, _ := resource.Merge(lambdaResource, customResource)
    xrayUdpExporter, _ := xrayudp.NewSpanExporter(ctx)
    tp := trace.NewTracerProvider(
        trace.WithSpanProcessor(trace.NewSimpleSpanProcessor(xrayUdpExporter)),
        trace.WithResource(mergedResource),
    )

    defer func(ctx context.Context) {
        err := tp.Shutdown(ctx)
        if err != nil {
            fmt.Printf("error shutting down tracer provider: %v", err)
        }
    }(ctx)

    otel.SetTracerProvider(tp)
    otel.SetTextMapPropagator(xray.Propagator{})

    lambda.Start(otellambda.InstrumentHandler(lambdaHandler,
 xrayconfig.WithRecommendedOptions(tp)...))
}
```

When invoking Lambda, you will see the following trace in the `Trace Map` in the CloudWatch console:



Migrate to OpenTelemetry Node.js

This section explains how to migrate your Node.js applications from X-Ray SDK to OpenTelemetry. It covers both automatic and manual instrumentation approaches, and provides specific examples for common use cases.

The X-Ray Node.js SDK helps you manually instrument your Node.js applications for tracing. This section provides code examples for migrating from X-Ray to OpenTelemetry instrumentation.

Sections

- Zero code automatic instrumentation solutions
- Manual instrumentation solutions
- Tracing incoming requests
- AWS SDK JavaScript V3 instrumentation
- Instrumenting outgoing HTTP calls
- Instrumentation support for other libraries
- Manually creating trace data
- Lambda instrumentation

Zero code automatic instrumentation solutions

To trace requests with the X-Ray SDK for Node.js, you must modify your application code. With OpenTelemetry, you can use zero-code auto-instrumentation solutions to trace requests.

Zero code automatic instrumentation with OpenTelemetry-based automatic instrumentations.

1. Using the AWS Distro for OpenTelemetry (ADOT) auto-instrumentation for Node.js – For automatic instrumentation for Node.js application, see Tracing and Metrics with the AWS Distro for OpenTelemetry JavaScript Auto-Instrumentation.

   (Optional) You can also enable CloudWatch Application Signals when automatically instrumenting your applications on AWS with the ADOT JavaScript auto-instrumentation to monitor current application health and track long-term application performance against your business objectives. Application Signals provides you with a unified, application-centric view of your applications, services, and dependencies, and helps you monitor and triage application health. For more information, see Application Signals.

2. Using the OpenTelemetry JavaScript zero-code automatic instrumentation – For automatic instrumentation with the OpenTelemetry JavaScript, see JavaScript zero-code instrumentation .

Manual instrumentation solutions

Tracing setup with X-Ray SDK

When the X-Ray SDK for Node.js was used, the `aws-xray-sdk` package was required to configure the X-Ray SDK with service plug-ins or local sampling rules before using the SDK to instrument your code.

```
var AWSXRay = require('aws-xray-sdk');

AWSXRay.config([AWSXRay.plugins.EC2Plugin,AWSXRay.plugins.ElasticBeanstalkPlugin]);
AWSXRay.middleware.setSamplingRules(<path to file>);
```

Tracing setup with OpenTelemetry SDK

> **Note**
>
> AWS X-Ray Remote Sampling is currently not available to be configured for OpenTelemetry JS. However, support for X-Ray Remote Sampling is currently available through the ADOT Auto-Instrumentation for Node.js.

For the code example below, you will need the following dependencies:

```
npm install --save \
  @opentelemetry/api \
  @opentelemetry/sdk-node \
  @opentelemetry/exporter-trace-otlp-proto \
  @opentelemetry/propagator-aws-xray \
  @opentelemetry/resource-detector-aws
```

You must set up and configure the OpenTelemetry SDK before running your application code. This can be done by using the --require flag. Create a file named *instrumentation.js*, which will contain your OpenTelemetry instrumentation configuration and setup.

It is recommend that you configure the following components:

- OTLPTraceExporter – Required for exporting traces to the CloudWatch Agent/OpenTelemetry Collector
- AWSXRayPropagator – Required for propagating the Trace Context to AWS Services that are integrated with X-Ray
- Resource Detectors (for example, Amazon EC2 Resource Detector) - To detect metadata of the host running your application

```
/*instrumentation.js*/
// Require dependencies
const { NodeSDK } = require('@opentelemetry/sdk-node');
const { OTLPTraceExporter } = require('@opentelemetry/exporter-trace-otlp-proto');
const { AWSXRayPropagator } = require("@opentelemetry/propagator-aws-xray");
const { detectResources } = require('@opentelemetry/resources');
const { awsEc2Detector } = require('@opentelemetry/resource-detector-aws');

const resource = detectResources({
    detectors: [awsEc2Detector],
});

const _traceExporter = new OTLPTraceExporter({
    url: 'http://localhost:4318/v1/traces'
});

const sdk = new NodeSDK({
    resource: resource,
    textMapPropagator: new AWSXRayPropagator(),
    traceExporter: _traceExporter
});

sdk.start();
```

Then, you can run your application with your OpenTelemetry setup like:

```
node --require ./instrumentation.js app.js
```

You can use OpenTelemetry SDK library instrumentations to automatically create spans for libraries such as the AWS SDK. Enabling these will automatically create spans for modules such as the AWS SDK for JavaScript v3. OpenTelemetry provides the option to enable all library instrumentations or specify which library instrumentations to enable.

To enable all instrumentations, install the `@opentelemetry/auto-instrumentations-node` package:

```
npm install @opentelemetry/auto-instrumentations-node
```

Next, update the configuration to enable all library instrumentations as shown below.

```
const { getNodeAutoInstrumentations } = require('@opentelemetry/auto-
instrumentations-node');

...

const sdk = new NodeSDK({
    resource: resource,
     instrumentations: [getNodeAutoInstrumentations()],
     textMapPropagator: new AWSXRayPropagator(),
    traceExporter: _traceExporter
});
```

Tracing setup with ADOT auto-instrumentation for Node.js

You can use the ADOT auto-instrumentation for Node.js to automatically configure OpenTelemetry for your Node.js applications. By using ADOT Auto-Instrumentation, you do not need to make manual code changes to trace incoming requests, or to trace libraries such as the AWS SDK or HTTP clients. For more information, see Tracing and metrics with the AWS Distro for OpenTelemetry JavaScript Auto-Instrumentation.

ADOT auto-instrumentation for Node.js supports:

- X-Ray remote sampling through environment variable – `export OTEL_TRACES_SAMPLER=xray`
- X-Ray trace context propagation (enabled by default)
- Resource detection (resource detection for Amazon EC2, Amazon ECS, and Amazon EKS environments are enabled by default)
- Automatic library instrumentations for all supported OpenTelemetry instrumentations, which can be disabled/enabled selectively through `OTEL_NODE_ENABLED_INSTRUMENTATIONS` and `OTEL_NODE_DISABLED_INSTRUMENTATIONS` environment variables
- Manual creation of Spans

Tracing incoming requests

With X-Ray SDK

Express.js

With the X-Ray SDK to trace incoming HTTP requests received by *Express.js* applications, the two middlewares `AWSXRay.express.openSegment(<name>)` and `AWSXRay.express.closeSegment()` were required to wrap all of your defined routes in order to trace them.

```
app.use(xrayExpress.openSegment('defaultName'));

...

app.use(xrayExpress.closeSegment());
```

Restify

To trace incoming HTTP requests received by `Restify` applications, the middleware from the X-Ray SDK was used by running enable from the `aws-xray-sdk-restify` module on the Restify Server:

```
var AWSXRay = require('aws-xray-sdk');
var AWSXRayRestify = require('aws-xray-sdk-restify');
var restify = require('restify');
var server = restify.createServer();
AWSXRayRestify.enable(server, 'MyApp'));
```

With OpenTelemetry SDK

Express.js

Tracing support for incoming requests for `Express.js` is provided by the OpenTelemetry HTTP instrumentation and OpenTelemetry express instrumentation. Install the following dependencies with npm:

```
npm install --save @opentelemetry/instrumentation-http @opentelemetry/
instrumentation-express
```

Update the OpenTelemetry SDK Configuration to enable instrumentation for the express module:

```
const { HttpInstrumentation } = require('@opentelemetry/instrumentation-http');
const { ExpressInstrumentation } = require('@opentelemetry/instrumentation-
express');
...

const sdk = new NodeSDK({
  ...

  instrumentations: [
    ...
    // Express instrumentation requires HTTP instrumentation
    new HttpInstrumentation(),
    new ExpressInstrumentation(),
  ],
});
```

Restify

For Restify applications, you will need the OpenTelemetry Restify instrumentation. Install the following dependency:

```
npm install --save @opentelemetry/instrumentation-restify
```

Update the OpenTelemetry SDK Configuration to enable instrumentation for the restify module:

```
const { RestifyInstrumentation } = require('@opentelemetry/instrumentation-
restify');
...

const sdk = new NodeSDK({
  ...

  instrumentations: [
    ...
    new RestifyInstrumentation(),
  ],
});
```

AWS SDK JavaScript V3 instrumentation

With X-Ray SDK

To instrument outgoing AWS requests from AWS SDK, you instrumented clients like the following example:

```
import { S3, PutObjectCommand } from '@aws-sdk/client-s3';

const s3 = AWSXRay.captureAWSv3Client(new S3({}));

await s3.send(new PutObjectCommand({
  Bucket: bucketName,
  Key: keyName,
  Body: 'Hello!',
}));
```

With OpenTelemetry SDK

Tracing support for downstream AWS SDK calls to DynamoDB, Amazon S3, and others is provided by the OpenTelemetry AWS SDK instrumentation. Install the following dependency with npm:

```
npm install --save @opentelemetry/instrumentation-aws-sdk
```

Update the OpenTelemetry SDK Configuration with the AWS SDK instrumentation.

```
import { AwsInstrumentation } from '@opentelemetry/instrumentation-aws-sdk';
...

const sdk = new NodeSDK({
  ...

  instrumentations: [
    ...
    new AwsInstrumentation()
  ],
});
```

Instrumenting outgoing HTTP calls

With X-Ray SDK

To instrument outgoing HTTP requests with X-Ray, it was required to instrument clients. For example, see below.

Individual HTTP clients

```
var AWSXRay = require('aws-xray-sdk');
var http = AWSXRay.captureHTTPs(require('http'));
```

All HTTP clients (global)

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.captureHTTPsGlobal(require('http'));
var http = require('http');
```

With OpenTelemetry SDK

Tracing support for Node.js HTTP clients is provided by the OpenTelemetry HTTP Instrumentation. Install the following dependency with npm:

```
npm install --save @opentelemetry/instrumentation-http
```

Update the OpenTelemetry SDK Configuration as follows:

```
const { HttpInstrumentation } = require('@opentelemetry/instrumentation-http');
...

const sdk = new NodeSDK({
  ...

  instrumentations: [
    ...
    new HttpInstrumentation(),
  ],
});
```

Instrumentation support for other libraries

You can find the full list of supported library instrumentations for OpenTelemetry JavaScript under Supported instrumentations .

Alternatively, you can search the OpenTelemetry registry to find out if OpenTelemetry supports instrumentation for your library under Registry.

Manually creating trace data

With X-Ray SDK

Using X-Ray, the `aws-xray-sdk` package code was required to manually create segments and their child sub-segments to trace your application.

```
var AWSXRay = require('aws-xray-sdk');

AWSXRay.enableManualMode();

var segment = new AWSXRay.Segment('myApplication');

captureFunc('1', function(subsegment1) {
  captureFunc('2', function(subsegment2) {

  }, subsegment1);
}, segment);

segment.close();
segment.flush();
```

With OpenTelemetry SDK

You can create and use custom spans to monitor the performance of internal activities that are not captured by instrumentation libraries. Note that only spans of kind Server are converted into X-Ray segments, all other spans are converted into X-Ray sub-segments. For more information, see Segments.

You will need a Tracer instance after you have configured the OpenTelemetry SDK in Tracing Setup to create Spans. You can create as many Tracer instances as needed, but it is common to have one Tracer for an entire application.

```
const { trace, SpanKind } = require('@opentelemetry/api');
```

```
// Get a tracer instance
const tracer = trace.getTracer('your-tracer-name');

...

  // This span will appear as a segment in X-Ray
  tracer.startActiveSpan('server', { kind: SpanKind.SERVER }, span => {
    // Do work here

    // This span will appear as a subsegment in X-Ray
    tracer.startActiveSpan('operation2', { kind: SpanKind.INTERNAL }, innerSpan => {
      // Do more work here

      innerSpan.end();
    });
    span.end();
  });
```

Adding annotations and metadata to traces with OpenTelemetry SDK

You can also add custom key-value pairs as attributes in your spans. Note that by default, all these span attributes will be converted into metadata in X-Ray raw data. To ensure that an attribute is converted into an annotation and not metadata, add the attribute's key to the list of the `aws.xray.annotations` attribute. For more information, see Enable The Customized X-Ray Annotations.

```

  tracer.startActiveSpan('server', { kind: SpanKind.SERVER }, span => {
    span.setAttribute('metadataKey', 'metadataValue');
    span.setAttribute('annotationKey', 'annotationValue');

    // The following ensures that "annotationKey: annotationValue" is an annotation
 in X-Ray raw data.
    span.setAttribute('aws.xray.annotations', ['annotationKey']);

    // Do work here

    span.end();
  });
```

Lambda instrumentation

With X-Ray SDK

After you enable *Active Tracing* for your Lambda function, the X-Ray SDK was required without additional configuration. Lambda creates a segment representing the Lambda handler invocation, and you created sub-segments or instrument libraries using the X-Ray SDK without any additional configuration.

With OpenTelemetry SDK

You can automatically instrument your Lambda with AWS vended Lambda layers. There are two solutions:

- (Recommended) CloudWatch Application Signals lambda layer

> **Note**
>
> This Lambda layer has CloudWatch Application Signals enabled by default, which enables performance and health monitoring for your Lambda application by collecting both metrics and traces. If you only want tracing, you should set the Lambda environment variable `OTEL_AWS_APPLICATION_SIGNALS_ENABLED=false`. For more information, see Enable your applications on Lambda .

- AWS managed Lambda layer for ADOT JS. For more information, see AWS Distro for OpenTelemetry Lambda Support For JavaScript.

Manually creating Spans with Lambda instrumentation

While the ADOT JavaScript Lambda Layer provides automatic instrumentation for your Lambda function, you might find the need to perform manual instrumentation in your Lambda, for example, to provide custom data or to instrument code within the Lambda function itself that is not covered by library instrumentations.

To perform manual instrumentation alongside the automatic instrumentation, you will need to add `@opentelemetry/api` as a dependency. The version of this dependency is recommended to be the same version of the same dependency that is used by the ADOT JavaScript SDK. You can use the OpenTelemetry API to manually create spans in your Lambda function.

To add the `@opentelemetry/api` dependency using NPM:

```
npm install @opentelemetry/api
```

Migrate to OpenTelemetry .NET

When using X-Ray Tracing in your .NET applications, the X-Ray .NET SDK with manual efforts is used for instrumentation.

This section provides code examples in the Manual instrumentation solutions with the SDK section for migrating from the X-Ray manual instrumentation solution to OpenTelemetry manual Instrumentation solutions for .NET. Alternatively, you can migrate from X-Ray manual instrumentation to OpenTelemetry automatic instrumentation solutions to instrument .NET applications without having to modify application source code in the Zero code automatic instrumentation solutions section.

Sections

- Zero code automatic instrumentation solutions
- Manual instrumentation solutions with the SDK
- Manually creating trace data
- Tracing incoming requests (ASP.NET and ASP.NET core instrumentation)
- AWS SDK instrumentation
- Instrumenting outgoing HTTP calls
- Instrumentation support for other libraries
- Lambda instrumentation

Zero code automatic instrumentation solutions

OpenTelemetry provides zero-code auto-instrumentation solutions. These solutions trace requests without requiring changes to your application code.

OpenTelemetry-based automatic instrumentation options

1. Using the AWS Distro for OpenTelemetry (ADOT) auto-Instrumentation for .NET – To automatically instrument .NET applications, see Tracing and Metrics with the AWS Distro for OpenTelemetry .NET Auto-Instrumentation.

   (Optional) Enable CloudWatch Application Signals when automatically instrumenting your applications on AWS with the ADOT .NET auto-instrumentation to:

   - Monitor current application health
   - Track long-term application performance against business objectives
   - Get a unified, application-centric view of your applications, services, and dependencies
   - Monitor and triage application health

   For more information, see Application Signals.
2. Using the OpenTelemetry .Net zero-code automatic instrumentation – To automatically instrument with OpenTelemetry .Net, see Tracing and Metrics with the AWS Distro for OpenTelemetry .NET Auto-Instrumentation.

Manual instrumentation solutions with the SDK

Tracing configuration with X-Ray SDK

For .NET web applications, the X-Ray SDK is configured in the appSettings section of the `Web.config` file.

Example Web.config

```
<configuration>
  <appSettings>
    <add key="AWSXRayPlugins" value="EC2Plugin"/>
  </appSettings>
</configuration>
```

For .NET Core, a file named `appsettings.json` with a top-level key named `XRay` is used, and then a configuration object is built o initialize the X-Ray recorder.

Example for .NET `appsettings.json`

```
{
  "XRay": {
    "AWSXRayPlugins": "EC2Plugin"
  }
}
```

Example for .NET Core Program.cs – Recorder configuration

```
using Amazon.XRay.Recorder.Core;
...
AWSXRayRecorder.InitializeInstance(configuration);
```

Tracing configuration with OpenTelemetry SDK

Add these dependencies:

```
dotnet add package OpenTelemetry
dotnet add package OpenTelemetry.Contrib.Extensions.AWSXRay
dotnet add package OpenTelemetry.Sampler.AWS --prerelease
dotnet add package OpenTelemetry.Resources.AWS
dotnet add package OpenTelemetry.Exporter.OpenTelemetryProtocol
dotnet add package OpenTelemetry.Extensions.Hosting
dotnet add package OpenTelemetry.Instrumentation.AspNetCore
```

For your .NET application, configure the OpenTelemetry SDK by setting up the Global TracerProvider. The following example configuration also enables instrumentation for `ASP.NET Core`. To instrument `ASP.NET`, see [Tracing incoming requests (ASP.NET and ASP.NET core instrumentation)](). To use OpenTelemetry with other frameworks, see [Registry]() for more libraries for supported frameworks.

It is recommend that you configure the following components:

- `An OTLP Exporter` – Required for exporting traces to the CloudWatch Agent/ OpenTelemetry Collector
- An AWS X-Ray Propagator – Required for propagating the Trace Context to [AWS Services that are integrated with X-Ray]()

- An AWS X-Ray Remote Sampler – Required if you need to sample requests using X-Ray sampling rules
- `Resource Detectors` (for example, Amazon EC2 Resource Detector) - To detect metadata of the host running your application

```
using OpenTelemetry;
using OpenTelemetry.Contrib.Extensions.AWSXRay.Trace;
using OpenTelemetry.Sampler.AWS;
using OpenTelemetry.Trace;
using OpenTelemetry.Resources;

var builder = WebApplication.CreateBuilder(args);

var serviceName = "MyServiceName";
var serviceVersion = "1.0.0";

var resourceBuilder = ResourceBuilder
    .CreateDefault()
    .AddService(serviceName: serviceName)
    .AddAWSEC2Detector();

builder.Services.AddOpenTelemetry()
    .ConfigureResource(resource => resource
        .AddAWSEC2Detector()
        .AddService(
            serviceName: serviceName,
            serviceVersion: serviceVersion))
    .WithTracing(tracing => tracing
        .AddSource(serviceName)
        .AddAspNetCoreInstrumentation()
        .AddOtlpExporter()
        .SetSampler(AWSXRayRemoteSampler.Builder(resourceBuilder.Build())
            .SetEndpoint("http://localhost:2000")
            .Build()));

Sdk.SetDefaultTextMapPropagator(new AWSXRayPropagator()); // configure  X-Ray
 propagator
```

To use OpenTelemetry for a console app, add the following OpenTelemetry configuration at the startup of your program.

```
using OpenTelemetry;
using OpenTelemetry.Contrib.Extensions.AWSXRay.Trace;
using OpenTelemetry.Trace;
using OpenTelemetry.Resources;

var serviceName = "MyServiceName";

var resourceBuilder = ResourceBuilder
    .CreateDefault()
    .AddService(serviceName: serviceName)
    .AddAWSEC2Detector();

var tracerProvider = Sdk.CreateTracerProviderBuilder()
    .AddSource(serviceName)
    .ConfigureResource(resource =>
        resource
            .AddAWSEC2Detector()
            .AddService(
                serviceName: serviceName,
                serviceVersion: serviceVersion
            )
        )
    .AddOtlpExporter() // default address localhost:4317
    .SetSampler(new TraceIdRatioBasedSampler(1.00))
    .Build();

Sdk.SetDefaultTextMapPropagator(new AWSXRayPropagator()); // configure  X-Ray
 propagator
```

Manually creating trace data

With X-Ray SDK

With the X-Ray SDK, the `BeginSegment` and `BeginSubsegment` methods were needed to manually create X-Ray segments and sub-segments.

```
using Amazon.XRay.Recorder.Core;

AWSXRayRecorder.Instance.BeginSegment("segment name"); // generates `TraceId` for
 you
try
{
    // Do something here
    // can create custom subsegments
    AWSXRayRecorder.Instance.BeginSubsegment("subsegment name");
    try
    {
        DoSometing();
    }
    catch (Exception e)
    {
        AWSXRayRecorder.Instance.AddException(e);
    }
    finally
    {
        AWSXRayRecorder.Instance.EndSubsegment();
    }
}
catch (Exception e)
{
    AWSXRayRecorder.Instance.AddException(e);
}
finally
{
    AWSXRayRecorder.Instance.EndSegment();
}
```

With OpenTelemetry SDK

In .NET, you can use the activity API to create custom spans to monitor the performance of internal activities that are not captured by instrumentation libraries. Note that only spans of kind Server are converted into X-Ray segments, all other spans are converted into X-Ray sub-egments.

You can create as many `ActivitySource` instances as needed, but it is recommended to have only one for an entire application/service.

```
using System.Diagnostics;

ActivitySource activitySource = new ActivitySource("ActivitySourceName",
 "ActivitySourceVersion");


...


using (var activity = activitySource.StartActivity("ActivityName",
 ActivityKind.Server)) // this will be translated to a X-Ray Segment
{
    // Do something here

    using (var internalActivity = activitySource.StartActivity("ActivityName",
 ActivityKind.Internal)) // this will be translated to an X-Ray Subsegment
    {
        // Do something here
    }
}
```

Adding annotations and metadata to traces with OpenTelemetry SDK

You can also add custom key-value pairs as attributes onto your spans by using the `SetTag` method on an activity. Note that by default, all the span attributes will be converted into metadata in X-Ray raw data. To ensure that an attribute is converted into an annotation and not metadata, you can add that attribute's key to the list of `aws.xray.annotations` attribute.

```
using (var activity = activitySource.StartActivity("ActivityName",
 ActivityKind.Server)) // this will be translated to a X-Ray Segment
{
    activity.SetTag("metadataKey", "metadataValue");
    activity.SetTag("annotationKey", "annotationValue");
    string[] annotationKeys = {"annotationKey"};
    activity.SetTag("aws.xray.annotations", annotationKeys);

    // Do something here
    using (var internalActivity = activitySource.StartActivity("ActivityName",
 ActivityKind.Internal)) // this will be translated to an X-Ray Subsegment
    {
        // Do something here
    }
}
```

With OpenTelemetry automatic instrumentation

If you are using an OpenTelemetry automatic instrumentation solution for .NET, and if you need to perform manual instrumentation in your application, for example, to instrument code within the application itself for sections that are not covered by any auto-instrumentation library.

Since there can only be one global `TracerProvider`, manual instrumentation should not instantiate its own `TracerProvider` if used together alongside auto-instrumentation. When `TracerProvider` is used, custom manual tracing works the same way when using automatic instrumentation or manual instrumentation through the OpenTelemetry SDK.

Tracing incoming requests (ASP.NET and ASP.NET core instrumentation)

With X-Ray SDK

To instrument requests served by the ASP.NET application, see https://docs.aws.amazon.com/xray/latest/devguide/xray-sdk-dotnet-messagehandler.html for information on how to call `RegisterXRay` in the `Init` method of your `global.asax` file.

```
AWSXRayASPNET.RegisterXRay(this, "MyApp");
```

To instrument requests served by your ASP.NET core application, the `UseXRay` method is called before any other middleware in the `Configure` method of your Startup class.

```
app.UseXRay("MyApp");
```

With OpenTelemetry SDK

OpenTelemetry also provides instrumentation libraries to collect traces for incoming web requests for ASP.NET and ASP.NET core. The following section lists the steps needed to add and enable these library instrumentations for your OpenTelemetry configuration, including how to add ASP.NET or ASP.NET core instrumentation when creating the Tracer Provider.

For information on how to enable OpenTelemetry.Instrumentation.AspNet, see Steps to enable OpenTelemetry.Instrumentation.AspNet and for information on how to enable OpenTelemetry.Instrumentation.AspNetCore, see Steps to enable OpenTelemetry.Instrumentation.AspNetCore .

AWS SDK instrumentation

With X-Ray SDK

Install all AWS SDK clients by calling `RegisterXRayForAllServices()`.

```
using Amazon.XRay.Recorder.Handlers.AwsSdk;
AWSSDKHandler.RegisterXRayForAllServices(); //place this before any instantiation of
 AmazonServiceClient
AmazonDynamoDBClient client = new AmazonDynamoDBClient(RegionEndpoint.USWest2); //
 AmazonDynamoDBClient is automatically registered with X-Ray
```

Use one of the following methods for specific AWS service client instrumentation.

```
AWSSDKHandler.RegisterXRay<IAmazonDynamoDB>(); // Registers specific type of
 AmazonServiceClient : All instances of IAmazonDynamoDB created after this line are
 registered
AWSSDKHandler.RegisterXRayManifest(String path); // To configure custom AWS Service
 Manifest file. This is optional, if you have followed "Configuration" section
```

With OpenTelemetry SDK

For the following code example, you will need the following dependency:

```
dotnet add package OpenTelemetry.Instrumentation.AWS
```

To instrument the AWS SDK, update the OpenTelemetry SDK configuration where the Global TracerProvider is setup.

```
builder.Services.AddOpenTelemetry()
    ...
    .WithTracing(tracing => tracing
        .AddAWSInstrumentation()
        ...
```

Instrumenting outgoing HTTP calls

With X-Ray SDK

The X-Ray .NET SDK traces outgoing HTTP calls through the extension methods `GetResponseTraced()` or `GetAsyncResponseTraced()` when using `System.Net.HttpWebRequest`, or by using the `HttpClientXRayTracingHandler` handler when using `System.Net.Http.HttpClient`.

With OpenTelemetry SDK

For the following code example, you will need the following dependency:

```
dotnet add package OpenTelemetry.Instrumentation.Http
```

To instrument `System.Net.Http.HttpClient` and `System.Net.HttpWebRequest`, update the OpenTelemetry SDK configuration where the Global TracerProvider is setup.

```
builder.Services.AddOpenTelemetry()
    ...
    .WithTracing(tracing => tracing
        .AddHttpClientInstrumentation()
        ...
```

Instrumentation support for other libraries

You can search and filter the OpenTelemetry Registry for .NET Instrumentation Libraries to find out if OpenTelemetry supports instrumentation for your Library. See the Registry to start searching.

Lambda instrumentation

With X-Ray SDK

The following procedure was required to use the X-Ray SDK with Lambda:

1. Enable *Active Tracing* on your Lambda function
2. The Lambda service creates a segment that represents your handler's invocation
3. Create sub-segments or instrument libraries using the X-Ray SDK

With OpenTelemetry-based solutions

You can automatically instrument your Lambda with AWS vended Lambda layers. There are two solutions:

- (Recommended) CloudWatch Application Signals lambda layer
- For better performance, you may want to consider using `OpenTelemetry Manual Instrumentation` to generate OpenTelemetry traces for your Lambda function.

OpenTelemetry manual instrumentation for AWS Lambda

The following is the Lambda function code (without instrumentation) example.

```
using System;
using System.Text;
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using Amazon.S3.Model;

// Assembly attribute to enable Lambda function logging
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace ExampleLambda;

public class ListBucketsHandler
{
    private static readonly AmazonS3Client s3Client = new();

    // new Lambda function handler passed in
    public async Task<string> HandleRequest(object input, ILambdaContext context)
    {
        try
        {
            var DoListBucketsAsyncResponse = await DoListBucketsAsync();
            context.Logger.LogInformation($"Results:
 {DoListBucketsAsyncResponse.Buckets}");

            context.Logger.LogInformation($"Successfully called ListBucketsAsync");
            return "Success!";
        }
        catch (Exception ex)
        {
            context.Logger.LogError($"Failed to call ListBucketsAsync: {ex.Message}");
            throw;
        }
    }

    private async Task<ListBucketsResponse> DoListBucketsAsync()
    {
        try
        {
            var putRequest = new ListBucketsRequest
            {
            };

            var response = await s3Client.ListBucketsAsync(putRequest);
            return response;
        }
        catch (AmazonS3Exception ex)
        {
            throw new Exception($"Failed to call ListBucketsAsync: {ex.Message}", ex);
        }
    }
}
```

To manually instrument your Lambda handler and the Amazon S3 client, do the following.

1. Instantiate a TracerProvider – The TracerProvider is recommended to be configured with an `XrayUdpSpanExporter`, a ParentBased Always On Sampler, and a `Resource` with `service.name` set to the Lambda function name.
2. Instrument the Amazon S3 client with the OpenTemetry AWS SDK instrumentation by calling `AddAWSInstrumentation()` to add AWS SDK client instrumentation to `TracerProvider`
3. Create a wrapper function with the same signature as the original Lambda function. Call `AWSLambdaWrapper.Trace()` API and pass `TracerProvider`, the original Lambda function, and its inputs as parameters. Set the wrapper function as the Lambda handler input.

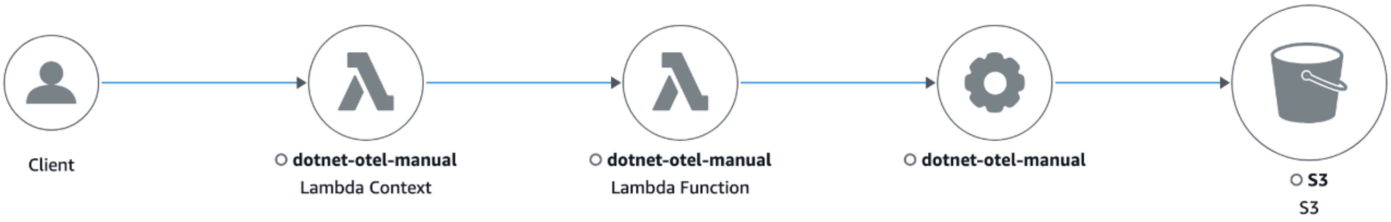
For the following code example, you will need the following dependencies:

```
dotnet add package OpenTelemetry.Instrumentation.AWSLambda
dotnet add package OpenTelemetry.Instrumentation.AWS
dotnet add package OpenTelemetry.Resources.AWS
dotnet add package AWS.Distro.OpenTelemetry.Exporter.Xray.Udp
```

The following code demonstrates the Lambda function after the required changes. You can create additional custom spans to complement the automatically provided spans.

```
using Amazon.Lambda.Core;
using Amazon.S3;
using Amazon.S3.Model;
using OpenTelemetry;
using OpenTelemetry.Instrumentation.AWSLambda;
using OpenTelemetry.Trace;
using AWS.Distro.OpenTelemetry.Exporter.Xray.Udp;
using OpenTelemetry.Resources;

// Assembly attribute to enable Lambda function logging
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer
namespace ExampleLambda;

public class ListBucketsHandler
{
    private static readonly AmazonS3Client s3Client = new();

    TracerProvider tracerProvider = Sdk.CreateTracerProviderBuilder()
        .AddAWSLambdaConfigurations()
        .AddProcessor(
            new SimpleActivityExportProcessor(
                // AWS_LAMBDA_FUNCTION_NAME Environment Variable will be defined in AWS
 Lambda Environment
                new
 XrayUdpExporter(ResourceBuilder.CreateDefault().AddService(Environment.GetEnvironmentVariable(
            )
        )
        .AddAWSInstrumentation()
        .SetSampler(new ParentBasedSampler(new AlwaysOnSampler()))
        .Build();

    // new Lambda function handler passed in
    public async Task<string> HandleRequest(object input, ILambdaContext context)
    => await AWSLambdaWrapper.Trace(tracerProvider, OriginalHandleRequest, input,
 context);

    public async Task<string> OriginalHandleRequest(object input, ILambdaContext
 context)
    {
        try
        {
            var DoListBucketsAsyncResponse = await DoListBucketsAsync();
            context.Logger.LogInformation($"Results:
 {DoListBucketsAsyncResponse.Buckets}");

            context.Logger.LogInformation($"Successfully called ListBucketsAsync");
            return "Success!";
        }
        catch (Exception ex)
        {
            context.Logger.LogError($"Failed to call ListBucketsAsync: {ex.Message}");
            throw;
        }
    }
    private async Task<ListBucketsResponse> DoListBucketsAsync()
    {
        try
        {
            var putRequest = new ListBucketsRequest
            {
            };

            var response = await s3Client.ListBucketsAsync(putRequest);
            return response;
        }
        catch (AmazonS3Exception ex)
        {
            throw new Exception($"Failed to call ListBucketsAsync: {ex.Message}", ex);
        }
    }
}
```

When invoking this Lambda, you will see the following trace in the Trace Map in the CloudWatch console:



Migrate to OpenTelemetry Python

This guide helps you migrate Python applications from X-Ray SDK to OpenTelemetry instrumentation. It covers both automatic and manual instrumentation approaches, with code examples for common scenarios.

Sections

- Zero code automatic instrumentation solutions
- Manually instrument your applications
- Tracing setup initialization

- Tracing incoming requests
- AWS SDK instrumentation
- Instrumenting outgoing HTTP calls through requests
- Instrumentation support for other libraries
- Manually creating trace data
- Lambda instrumentation

Zero code automatic instrumentation solutions

With X-Ray SDK, you had to modify your application code to trace requests. OpenTelemetry offers zero-code auto-instrumentation solutions to trace requests. With OpenTelemetry, you have the option of using zero-code auto-instrumentation solutions to trace requests.

Zero code with OpenTelemetry-based automatic instrumentations

1. Using the AWS Distro for OpenTelemetry (ADOT) auto-Instrumentation for Python – For automatic instrumentation for Python applications, see Tracing and Metrics with the AWS Distro for OpenTelemetry Python Auto-Instrumentation.

   (Optional) You can also enable CloudWatch Application Signals when automatically instrumenting your applications on AWS with the ADOT Python auto-instrumentation to monitor current application health and track long-term application performance against your business objectives. Application Signals provides you with a unified, application-centric view of your applications, services, and dependencies, and helps you monitor and triage application health.
2. Using the OpenTelemetry Python zero-code automatic instrumentation – For automatic instrumentation with the OpenTelemetry Python, see Python zero-code instrumentation.

Manually instrument your applications

You can manually instrument your applications using the `pip` command.

With X-Ray SDK

```
pip install aws-xray-sdk
```

With OpenTelemetry SDK

```
pip install opentelemetry-api opentelemetry-sdk opentelemetry-exporter-otlp
 opentelemetry-propagator-aws-xray
```

Tracing setup initialization

With X-Ray SDK

In X-Ray, the global `xray_recorder` is initialized and used it to generate segments and sub-segments.

With OpenTelemetry SDK

> **Note**
>
> X-Ray Remote Sampling is currently not available to be configured for OpenTelemetry Python. However, support for X-Ray Remote Sampling is currently available through the ADOT Auto-Instrumentation for Python.

In OpenTelemetry, you need to initialize a global `TracerProvider`. Using this `TracerProvider`, you can acquire a Tracer that you can use to generate spans anywhere in your application. It is recommend that you configure the following components:

- `OTLPSpanExporter` – Required for exporting traces to the CloudWatch Agent/OpenTelemetry Collector
- An AWS X-Ray Propagator – Required for propagating the Trace Context to AWS Services that are integrated with X-Ray

```
from opentelemetry import (
    trace,
    propagate
)
from opentelemetry.sdk.trace import TracerProvider

from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.exporter.otlp.proto.http.trace_exporter import OTLPSpanExporter
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.propagators.aws import AwsXRayPropagator

# Sends generated traces in the OTLP format to an OTel Collector running on port
 4318
otlp_exporter = OTLPSpanExporter(endpoint="http://localhost:4318/v1/traces")
# Processes traces in batches as opposed to immediately one after the other
span_processor = BatchSpanProcessor(otlp_exporter)
# More configurations can be done here. We will visit them later.

# Sets the global default tracer provider
provider = TracerProvider(active_span_processor=span_processor)
trace.set_tracer_provider(provider)

# Configures the global propagator to use the X-Ray Propagator
propagate.set_global_textmap(AwsXRayPropagator())

# Creates a tracer from the global tracer provider
tracer = trace.get_tracer("my.tracer.name")
# Use this tracer to create Spans
```

With ADOT auto-Instrumentation for Python

You can use the ADOT auto-instrumentation for Python to automatically configure OpenTelemetry for your Python applications. By using ADOT auto-instrumentation, you do not need to make manual code changes to trace incoming requests, or to trace libraries such as the AWS SDK or HTTP clients. For more information, see Tracing and Metrics with the AWS Distro for OpenTelemetry Python Auto-Instrumentation.

ADOT auto-instrumentation for Python supports:

- X-Ray remote sampling through the environment variable `export OTEL_TRACES_SAMPLER=xray`
- X-Ray trace context propagation (enabled by default)

- Resource detection (resource detection for Amazon EC2, Amazon ECS, and Amazon EKS environments are enabled by default)
- Automatic library instrumentations for all supported OpenTelemetry instrumentations are enabled by default. You can disable selectively through the `OTEL_PYTHON_DISABLED_INSTRUMENTATIONS` environment variable. (all are enabled by default)
- Manual creation of Spans

From X-Ray service plug-ins to OpenTelemetry AWS resource providers

The X-Ray SDK provides plug-ins that you could add to the `xray_recorder` to capture the platform specific information from the hosted service like Amazon EC2, Amazon ECS, and Elastic Beanstalk. It's similar to the Resource Providers in OpenTelemetry that captures the information as Resource attributes. There are multiple Resource Providers available for different AWS platforms.

- Start by installing the AWS extension package, `pip install opentelemetry-sdk-extension-aws`
- Configure the desired resource detector. The following example shows how to configure the Amazon EC2 resource provider in OpenTelemetry SDK

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.extension.aws.resource.ec2 import (
    AwsEc2ResourceDetector,
)
from opentelemetry.sdk.resources import get_aggregated_resources

provider = TracerProvider(
    active_span_processor=span_processor,
    resource=get_aggregated_resources([
        AwsEc2ResourceDetector(),
    ]))

trace.set_tracer_provider(provider)
```

Tracing incoming requests

With X-Ray SDK

The X-Ray Python SDK supports application frameworks like Django, Flask, and Bottle in tracing the incoming requests for Python applications running on them. This is done by adding `XRayMiddleware` to the application for each framework.

With OpenTelemetry SDK

OpenTelemetry provides instrumentations for Django and Flask through the specific instrumentation libraries. There is no instrumentation for Bottle available in OpenTelemetry, applications can still be traced by using the OpenTelemetry WSGI Instrumentation .

For the following code example, you need the following dependency:

```
pip install opentelemetry-instrumentation-flask
```

You must initialize the OpenTelemetry SDK and register the global TracerProvider before adding instrumentations for your application framework. Without it, the trace operations will be `no-ops`. Once you have configured the global `TracerProvider`, you can use the instrumentor for your application framework. The following example demonstrates a Flask application.

```
from flask import Flask
from opentelemetry import trace
from opentelemetry.instrumentation.flask import FlaskInstrumentor
from opentelemetry.sdk.extension.aws.resource import AwsEc2ResourceDetector
from opentelemetry.sdk.resources import get_aggregated_resources
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor, ConsoleSpanExporter

provider = TracerProvider(resource=get_aggregated_resources(
    [
        AwsEc2ResourceDetector(),
    ]))

processor = BatchSpanProcessor(ConsoleSpanExporter())
```

```
provider.add_span_processor(processor)
trace.set_tracer_provider(provider)

# Creates a tracer from the global tracer provider
tracer = trace.get_tracer("my.tracer.name")

app = Flask(__name__)

# Instrument the Flask app
FlaskInstrumentor().instrument_app(app)


@app.route('/')
def hello_world():
    return 'Hello World!'


if __name__ == '__main__':
    app.run()
```

AWS SDK instrumentation

With X-Ray SDK

The X-Ray Python SDK traces the AWS SDK client request by patching the `botocore` library. For more information, see [Tracing AWS SDK calls with the X-Ray SDK for Python](). In your application, the `patch_all()` method is used to instrument all the libraries or patch selectively using the `botocore` or `boto3`libraries using `patch((['botocore']))`. Any of the chosen method instruments all the Boto3 clients in your application and generates a sub-segment for any call made using these clients.

With OpenTelemetry SDK

For the following code example, you will need the following dependency:

```
pip install opentelemetry-instrumentation-botocore
```

Use the OpenTelemetry Botocore Instrumentation programmatically to instrument all the Boto3 clients in your application. The following example demonstrates the `botocore` instrumentation.

```
import boto3
import opentelemetry.trace as trace
from botocore.exceptions import ClientError
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.resources import get_aggregated_resources
from opentelemetry.sdk.trace.export import (
    BatchSpanProcessor,
    ConsoleSpanExporter,
)
from opentelemetry.instrumentation.botocore import BotocoreInstrumentor

provider = TracerProvider()
processor = BatchSpanProcessor(ConsoleSpanExporter())
provider.add_span_processor(processor)
trace.set_tracer_provider(provider)

# Creates a tracer from the global tracer provider
tracer = trace.get_tracer("my.tracer.name")

# Instrument BotoCore
BotocoreInstrumentor().instrument()

# Initialize S3 client
s3 = boto3.client("s3", region_name="us-east-1")

# Your bucket name
bucket_name = "my-example-bucket"

# Get bucket location (as an example of describing it)
try:
    response = s3.get_bucket_location(Bucket=bucket_name)
    region = response.get("LocationConstraint") or "us-east-1"
    print(f"Bucket '{bucket_name}' is in region: {region}")

    # Optionally, get bucket's creation date via list_buckets
    buckets = s3.list_buckets()
    for bucket in buckets["Buckets"]:
        if bucket["Name"] == bucket_name:
            print(f"Bucket created on: {bucket['CreationDate']}")
            break
except ClientError as e:
    print(f"Failed to describe bucket: {e}")
```

Instrumenting outgoing HTTP calls through requests

With X-Ray SDK

The X-Ray Python SDK traces outgoing HTTP calls through requests by patching the requests library. For more information, see Tracing calls to downstream HTTP web services using the X-Ray SDK for Python. In your application, you can use the `patch_all()` method to instrument all the libraries or by selectively patching the requests libraries by using `patch((['requests']))`. Any of the option instruments the `requests` library, generating a sub-segment for any call made through `requests`.

With OpenTelemetry SDK

For the following code example, you will need the following dependency:

```
pip install opentelemetry-instrumentation-requests
```

Use the OpenTelemetry Requests Instrumentation programmatically to instrument the requests library to generate traces for HTTP requests made by it in your application. For more information, see OpenTelemetry requests Instrumentation. The following example demonstrates the `requests` library instrumentation.

```
from opentelemetry.instrumentation.requests import RequestsInstrumentor

# Instrument Requests
RequestsInstrumentor().instrument()

...

    example_session = requests.Session()
    example_session.get(url="https://example.com")
```

Alternatively, you can also instrument the underlying `urllib3` library to trace HTTP requests:

```
# pip install opentelemetry-instrumentation-urllib3
from opentelemetry.instrumentation.urllib3 import URLLib3Instrumentor

# Instrument urllib3
URLLib3Instrumentor().instrument()

...

    example_session = requests.Session()
    example_session.get(url="https://example.com")
```

Instrumentation support for other libraries

You can find the full list of supported Library instrumentations for OpenTelemetry Python under Supported libraries, frameworks, application servers, and JVMs .

Alternatively, you can search the OpenTelemetry Registry to find out if OpenTelemetry supports instrumentation. See the Registry to start searching.

Manually creating trace data

You can create segments and sub-segments using the `xray_recorder` in your Python application. For more information, see Instrumenting Python code manually . You can also manually add annotations and metadata to the trace data.

Creating spans With OpenTelemetry SDK

Use the `start_as_current_span` API to start a span and set it for creating spans. For examples on creating spans, see Creating spans. Once a span is started and is in the current scope, you can add more information to it by adding attributes, events, exceptions, links, and so on. Like how we have segments and sub-segments in X-Ray, there are different kinds of spans in OpenTelemetry. Only the SERVER kind spans are converted to X-Ray segments while others are converted to X-Ray sub-segments.

```
from opentelemetry import trace
from opentelemetry.trace import SpanKind

import time

tracer = trace.get_tracer("my.tracer.name")

# Create a new span to track some work
with tracer.start_as_current_span("parent", kind=SpanKind.SERVER) as parent_span:
    time.sleep(1)

    # Create a nested span to track nested work
    with tracer.start_as_current_span("child", kind=SpanKind.CLIENT) as child_span:
        time.sleep(2)
        # the nested span is closed when it's out of scope

    # Now the parent span is the current span again
    time.sleep(1)

    # This span is also closed when it goes out of scope
```

Adding annotations and metadata to traces with OpenTelemetry SDK

The X-Ray Python SDK provides separate APIs, `put_annotation` and `put_metadata` for adding annotations and metadata to a trace. In OpenTelemetry SDK, the annotations and metadata are simply attributes on a span, added through the `set_attribute` API.

Span attributes that you want them to be annotations on a trace are added under the reserved key `aws.xray.annotations` whose value is a list of key-value pairs of annotations. All the other span attributes become metadata on the converted segment or sub-segment.

Additionally, if you are using the ADOT collector you can configure which span attributes should be converted to X-Ray annotations by specifying the `indexed_attributes` in the collector configuration.

The below example demonstrates how to add annotations and metadata to a trace using OpenTelemetry SDK.

```
with tracer.start_as_current_span("parent", kind=SpanKind.SERVER) as parent_span:
    parent_span.set_attribute("TransactionId", "qwerty12345")
    parent_span.set_attribute("AccountId", "1234567890")

    # This will convert the TransactionId and AccountId to be searchable X-Ray 
annotations
    parent_span.set_attribute("aws.xray.annotations", ["TransactionId", "AccountId"])

    with tracer.start_as_current_span("child", kind=SpanKind.CLIENT) as child_span:

        # The MicroTransactionId will be converted to X-Ray metadata for the child 
subsegment
        child_span.set_attribute("MicroTransactionId", "micro12345")
```

Lambda instrumentation

To monitor your lambda functions on X-Ray, you enable X-Ray and added appropriate permissions to the function invocation role. Additionally, if you are tracing downstream requests from your function, you would be instrumenting the code with X-Ray Python SDK.

With OpenTelemetry for X-Ray, it is recommended to use the CloudWatch Application Signals lambda layer with Application Signals turned off. This will auto-instrument your function and will generate spans for the function invocation and any downstream request from your function. Besides tracing, if you are interested in using Application Signals to monitor the health of your function, see Enable your applications on Lambda .

- Find the required Lambda layer ARN for your function from AWS Lambda Layer for OpenTelemetry ARNs and add it.
- Set the following environment variables for your function.
  - `AWS_LAMBDA_EXEC_WRAPPER=/opt/otel-instrument` – This loads the auto-instrumentation for the function
  - `OTEL_AWS_APPLICATION_SIGNALS_ENABLED=false` – This will disable Application Signals monitoring

Manually creating spans with Lambda instrumentation

Additionally, you can generate custom spans within your function to track work. You can do by using only the `opentelemetry-api` package in conjunction with the Application Signals lambda layer auto-instrumentation.

1. Include the `opentelemetry-api` as a dependency in your function
2. The following code snippet is a sample to generate custom spans

```
from opentelemetry import trace

# Get the tracer (auto-configured by the Application Signals layer)
tracer = trace.get_tracer(__name__)

def handler(event, context):
    # This span is a child of the layer's root span
    with tracer.start_as_current_span("my-custom-span") as span:
        span.set_attribute("key1", "value1")
        span.add_event("custom-event", {"detail": "something happened"})

        # Any logic you want to trace
        result = some_internal_logic()

    return {
        "statusCode": 200,
        "body": result
    }
```

Migrate to OpenTelemetry Ruby

To migrate your Ruby applications from X-Ray SDK to OpenTelemetry instrumentation, use the following code examples and guidance for manual instrumentation.

Sections

- Manually instrument your solutions with the SDK
- Tracing incoming requests (Rails instrumentation)
- AWS SDK instrumentation
- Instrumenting outgoing HTTP calls
- Instrumentation support for other libraries
- Manually creating trace data
- Lambda manual instrumentation

Manually instrument your solutions with the SDK

Tracing setup with X-Ray SDK

X-Ray SDK for Ruby required you to configure your code with service plug-ins.

```
require 'aws-xray-sdk'

XRay.recorder.configure(plugins: [:ec2, :elastic_beanstalk])
```

Tracing setup with OpenTelemetry SDK

> **Note**
>
> X-Ray remote sampling is currently not available to be configured for OpenTelemetry Ruby.

For a Ruby on Rails application, place your configuration code in a Rails initializer. For more information, see Getting Started. For all manually instrumented Ruby programs, you must use the `OpenTelemetry::SDK.configure` method to configure the OpenTelemetry Ruby SDK.

First, install the following packages:

```
bundle add opentelemetry-sdk opentelemetry-exporter-otlp opentelemetry-propagator-
xray
```

Next, configure the OpenTelemetry SDK through the configuration code that runs when your program initializes. It is recommend that you configure the following components:

- `OTLP Exporter` – Required for exporting traces to the CloudWatch agent and OpenTelemetry collector
- `An AWSX-Ray Propagator` – Required for propagating the trace context to AWS services that are integrated with X-Ray

```
require 'opentelemetry-sdk'
require 'opentelemetry-exporter-otlp'
```

```
# Import the gem containing the AWS X-Ray for OTel Ruby ID Generator and propagator
require 'opentelemetry-propagator-xray'

OpenTelemetry::SDK.configure do |c|
  c.service_name = 'my-service-name'

  c.add_span_processor(
    # Use the BatchSpanProcessor to send traces in groups instead of one at a time
    OpenTelemetry::SDK::Trace::Export::BatchSpanProcessor.new(
      # Use the default OLTP Exporter to send traces to the ADOT Collector
      OpenTelemetry::Exporter::OTLP::Exporter.new(
        # The OpenTelemetry Collector is running as a sidecar and listening on port
 4318
        endpoint:"http://127.0.0.1:4318/v1/traces"
      )
    )
  )

  # The X-Ray Propagator injects the X-Ray Tracing Header into downstream calls
  c.propagators = [OpenTelemetry::Propagator::XRay::TextMapPropagator.new]
end
```

OpenTelemetry SDKs also have the concept of library instrumentations. Enabling these will automatically create spans for libraries such as the AWS SDK. OpenTelemetry provides the option to enable all library instrumentations or specify which library instrumentations to enable.

To enable all instrumentations, first install the `opentelemetry-instrumentation-all` package:

```
bundle add opentelemetry-instrumentation-all
```

Next, update the configuration to enable all library instrumentations as shown below:

```
require 'opentelemetry/instrumentation/all'
...

OpenTelemetry::SDK.configure do |c|
   ...
```

```
  c.use_all() # Enable all instrumentations
end
```

OpenTelemetry SDKs also have the concept of library instrumentations. Enabling these will automatically create spans for libraries such as the AWS SDK. OpenTelemetry provides the option to enable all library instrumentations or specify which library instrumentations to enable.

To enable all instrumentations, first install the `opentelemetry-instrumentation-all` package:

```
bundle add opentelemetry-instrumentation-all
```

Next, update the configuration to enable all library instrumentations as shown below:

```
require 'opentelemetry/instrumentation/all'
...

OpenTelemetry::SDK.configure do |c|
    ...

  c.use_all() # Enable all instrumentations
end
```

Tracing incoming requests (Rails instrumentation)

With X-Ray SDK

With X-Ray SDK, X-Ray tracing is configured for the Rails framework upon initialization.

Example – config/initializers/aws_xray.rb

```
Rails.application.config.xray = {
  name: 'my app',
  patch: %I[net_http aws_sdk],
  active_record: true
}
```

With OpenTelemetry SDK

First, install the following packages:

```
bundle add opentelemetry-instrumentation-rack opentelemetry-instrumentation-
rails opentelemetry-instrumentation-action_pack opentelemetry-instrumentation-
active_record opentelemetry-instrumentation-action_view
```

Next, update the configuration to enable instrumentation for your Rails application as shown below:

```
# During SDK configuration
OpenTelemetry::SDK.configure do |c|

  ...

  c.use 'OpenTelemetry::Instrumentation::Rails'
  c.use 'OpenTelemetry::Instrumentation::Rack'
  c.use 'OpenTelemetry::Instrumentation::ActionPack'
  c.use 'OpenTelemetry::Instrumentation::ActiveSupport'
  c.use 'OpenTelemetry::Instrumentation::ActionView'

  ...

end
```

AWS SDK instrumentation

With X-Ray SDK

To instrument outgoing AWS requests from AWS SDK, the AWS SDK clients are patched with X-Ray like the following example:

```
require 'aws-xray-sdk'
require 'aws-sdk-s3'
```

```
# Patch AWS SDK clients
XRay.recorder.configure(plugins: [:aws_sdk])

# Use the instrumented client
s3 = Aws::S3::Client.new
s3.list_buckets
```

With OpenTelemetry SDK

AWS SDK for Ruby V3 provides support for recording and emitting OpenTelemetry traces. For information on how to configure OpenTelemetry for a service client, see Configuring observability features in the AWS SDK for Ruby .

Instrumenting outgoing HTTP calls

When making HTTP calls to external services, you might need to manually instrument the calls if automatic instrumentation isn't available or doesn't provide enough detail.

With X-Ray SDK

To instrument downstream calls, the X-Ray SDK for Ruby was used to patch the `net/http` library that your application uses:

```
require 'aws-xray-sdk'

config = {
  name: 'my app',
  patch: %I[net_http]
}

XRay.recorder.configure(config)
```

With OpenTelemetry SDK

To enable the `net/http` instrumentation using OpenTelemetry, first install the `opentelemetry-instrumentation-net_http` package:

```
bundle add opentelemetry-instrumentation-net_http
```

Next, update the configuration to enable the `net/http` instrumentation as shown below:

```
OpenTelemetry::SDK.configure do |c|
   ...

  c.use 'OpenTelemetry::Instrumentation::Net::HTTP'
  ...

end
```

Instrumentation support for other libraries

You can find the full list of supported Library instrumentations for OpenTelemetry Ruby under opentelemetry-ruby-contrib .

Alternatively, you can search the OpenTelemetry Registry to find out if OpenTelemetry supports instrumentation. For more information, see Registry.

Manually creating trace data

With X-Ray SDK

Using X-Ray, the `aws-xray-sdk` package required you to manually create segments and their child sub-segments to trace your application. You may have also added X-Ray annotations and metadata to your segments or sub-segments:

```
require 'aws-xray-sdk'
...

# Start a segment
segment = XRay.recorder.begin_segment('my-service')

# Add annotations (indexed key-value pairs)
segment.annotations[:user_id] = 'user-123'
segment.annotations[:payment_status] = 'completed'

# Add metadata (non-indexed data)
segment.metadata[:order] = {
  id: 'order-456',
  items: [
    { product_id: 'prod-1', quantity: 2 },
    { product_id: 'prod-2', quantity: 1 }
  ],
  total: 67.99
}

# Add metadata to a specific namespace
segment.metadata(namespace: 'payment') do |metadata|
  metadata[:transaction_id] = 'tx-789'
  metadata[:payment_method] = 'credit_card'
end

# Create a subsegment with annotations and metadata
segment.subsegment('payment-processing') do |subsegment1|
  subsegment1.annotations[:payment_id] = 'pay-123'
  subsegment1.metadata[:details] = { amount: 67.99, currency: 'USD' }

  # Create a nested subsegment
  subsegment1.subsegment('operation-2') do |subsegment2|
    # Do more work...
  end
end

# Close the segment
segment.close
```

With OpenTelemetry SDK

You can use custom spans to monitor the performance of internal activities that are not captured by instrumentation libraries. Note that only spans of kind server are converted into X-Ray segments, all other spans are converted into X-Ray sub-segments. By default, spans are `INTERNAL`.

First, create a Tracer in order to generate spans, which you can obtain through the `OpenTelemetry.tracer_provider.tracer('<YOUR_TRACER_NAME>')` method. This will provide a Tracer instance that is registered globally in you application's OpenTelemetry configuration. It is common to have a single Tracer for an entire application. Create an OpenTelemetry tracer and use it to create spans:

```
require 'opentelemetry-sdk'
```

```
...

# Get a tracer
tracer = OpenTelemetry.tracer_provider.tracer('my-application')

# Create a server span (equivalent to X-Ray segment)
tracer.in_span('my-application', kind: OpenTelemetry::Trace::SpanKind::SERVER) do |
span|
  # Do work...

  # Create nested spans of default kind INTERNAL will become an X-Ray subsegment
  tracer.in_span('operation-1') do |child_span1|
    # Set attributes (equivalent to X-Ray annotations and metadata)
    child_span1.set_attribute('key', 'value')

    # Do more work...
    tracer.in_span('operation-2') do |child_span2|
      # Do more work...
    end
  end
end
```

Adding annotations and metadata to traces with OpenTelemetry SDK

Use the `set_attribute` method to add attributes to each span. Note that by default, all these span attributes will be converted into metadata in X-Ray raw data. To ensure that an attribute is converted into an annotation and not metadata, you can add that attributes key to the list of `aws.xray.annotations` attribute. For more information, see Enable The Customized X-Ray Annotations .

```
# SERVER span will become an X-Ray segment
tracer.in_span('my-server-operation', kind: OpenTelemetry::Trace::SpanKind::SERVER)
 do |span|
    # Your server logic here
    span.set_attribute('attribute.key', 'attribute.value')
    span.set_attribute("metadataKey", "metadataValue")
    span.set_attribute("annotationKey1", "annotationValue")

    # Create X-Ray annotations
    span.set_attribute("aws.xray.annotations", ["annotationKey1"])
end
```

Lambda manual instrumentation

With X-Ray SDK

After *Active Tracing* was enabled on Lambda, there are no additional configurations required to use the X-Ray SDK. Lambda will create a segment representing the Lambda handler invocation, and you can create sub-segments or instrument libraries using the X-Ray SDK without any additional configuration.

With OpenTelemetry SDK

Consider the following sample Lambda function code (without instrumentation):

```
require 'json'
def lambda_handler(event:, context:)
    # TODO implement
    { statusCode: 200, body: JSON.generate('Hello from Lambda!') }
end
```

To manually instrument your Lambda, you will need to:

1. Add the following gems for your Lambda

   ```
   gem 'opentelemetry-sdk'
   gem 'opentelemetry-exporter-otlp'
   gem 'opentelemetry-propagator-xray'
   gem 'aws-distro-opentelemetry-exporter-xray-udp'
   gem 'opentelemetry-instrumentation-aws_lambda'
   gem 'opentelemetry-propagator-xray', '~> 0.24.0' # Requires version v0.24.0 or
    higher
   ```

2. Initialize OpenTelemetry SDK outside your Lambda Handler. The OpenTelemetry SDK is recommended to be configured with:

   1. A simple span processor with an X-Ray UDP span exporter to send Traces to Lambda's UDP X-Ray endpoint
   2. An X-Ray Lambda propagator

    3. `service_name` configuration to be set to the Lambda function name

3. In your Lambda handler class, add the following lines to instrument your Lambda Handler:

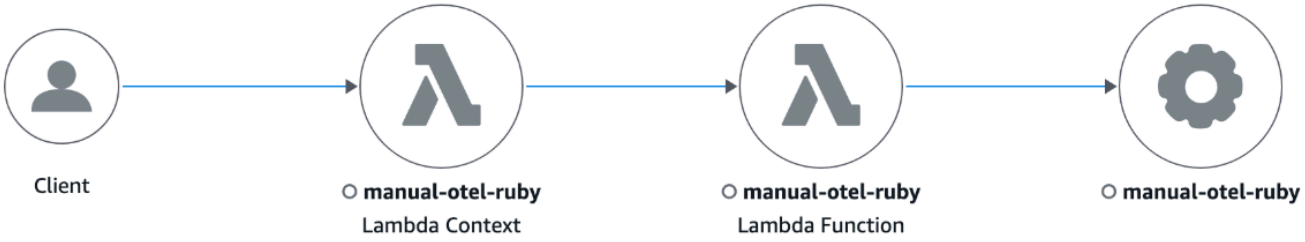
   ```
   class Handler
         extend OpenTelemetry::Instrumentation::AwsLambda::Wrap
         ...

         instrument_handler :process
     end
   ```

The following code demonstrates the Lambda function after the required changes. You can create additional custom spans to complement the automatically provided spans.

```
require 'json'
require 'opentelemetry-sdk'
require 'aws/distro/opentelemetry/exporter/xray/udp'
require 'opentelemetry/propagator/xray'
require 'opentelemetry/instrumentation/aws_lambda'

# Initialize OpenTelemetry SDK outside handler
OpenTelemetry::SDK.configure do |c|
  # Configure the AWS Distro for OpenTelemetry X-Ray Lambda exporter
  c.add_span_processor(
    OpenTelemetry::SDK::Trace::Export::SimpleSpanProcessor.new(
      AWS::Distro::OpenTelemetry::Exporter::XRay::UDP::AWSXRayUDPSpanExporter.new
    )
  )

  # Configure X-Ray Lambda propagator
  c.propagators = [OpenTelemetry::Propagator::XRay.lambda_text_map_propagator]

  # Set minimal resource information
  c.resource = OpenTelemetry::SDK::Resources::Resource.create({
    OpenTelemetry::SemanticConventions::Resource::SERVICE_NAME =>
 ENV['AWS_LAMBDA_FUNCTION_NAME']
  })
  c.use 'OpenTelemetry::Instrumentation::AwsLambda'
end

module LambdaFunctions
  class Handler
```

```
    extend OpenTelemetry::Instrumentation::AwsLambda::Wrap
    def self.process(event:, context:)
      "Hello!"
    end
    instrument_handler :process
  end
end
```

The following is an example trace map of an instrumented Lambda function written in Ruby.



You can also use Lambda layers to configure OpenTelemetry for your Lambda. For more information, see OpenTelemetry AWS-Lambda Instrumentation .

Creating X-Ray resources with AWS CloudFormation

AWS X-Ray is integrated with AWS CloudFormation, a service that helps you to model and set up your AWS resources so that you can spend less time creating and managing your resources and infrastructure. You create a template that describes all the AWS resources that you want, and AWS CloudFormation provisions and configures those resources for you.

When you use AWS CloudFormation, you can reuse your template to set up your X-Ray resources consistently and repeatedly. Describe your resources once, and then provision the same resources over and over in multiple AWS accounts and Regions.

X-Ray and AWS CloudFormation templates

To provision and configure resources for X-Ray and related services, you must understand AWS CloudFormation templates. Templates are formatted text files in JSON or YAML. These templates describe the resources that you want to provision in your AWS CloudFormation stacks. If you're unfamiliar with JSON or YAML, you can use AWS CloudFormation Designer to help you get started with AWS CloudFormation templates. For more information, see What is AWS CloudFormation Designer? in the *AWS CloudFormation User Guide*.

X-Ray supports creating `AWS::XRay::Group`, `AWS::XRay::SamplingRule`, and `AWS::XRay::ResourcePolicy` resources in AWS CloudFormation. For more information, including examples of JSON and YAML templates, see the X-Ray resource type reference in the *AWS CloudFormation User Guide*.

Learn more about AWS CloudFormation

To learn more about AWS CloudFormation, see the following resources:

- AWS CloudFormation
- AWS CloudFormation User Guide
- AWS CloudFormation API Reference
- AWS CloudFormation Command Line Interface User Guide

Tagging X-Ray sampling rules and groups

Tags are words or phrases that you can use to identify and organize your AWS resources. You can add multiple tags to each resource. Each tag includes a key and an optional value that you define. For example, a tag key might be **`domain`**, and the tag value might be **`example.com`**. You can search and filter your resources based on tags that you add. For more information about ways to use tags, see Tagging AWS resources in the *AWS General Reference*.

You can use tags to enforce tag-based permissions on CloudFront distributions. For more information, see Controlling Access to AWS Resources Using Resource Tags.

Note

Tag Editor and AWS Resource Groups do not currently support X-Ray resources. You add and manage tags by using the AWS X-Ray console or API.

You can apply tags to resources by using the X-Ray console, API, AWS CLI, SDKs, and AWS Tools for Windows PowerShell. For more information, see the following documentation:

- X-Ray API – See the following operations in the *AWS X-Ray API Reference*:
  - ListTagsForResource
  - CreateSamplingRule
  - CreateGroup
  - TagResource
  - UntagResource
- AWS CLI – See xray in the *AWS CLI Command Reference*
- SDKs – See the applicable SDK documentation on the AWS Documentation page

Note

If you cannot add or change tags on an X-Ray resource, or you cannot add a resource that has specific tags, you might not have permissions to perform this operation. To request access, contact an AWS user in your enterprise who has **Administrator** permissions in X-Ray.

Topics

- Tag restrictions
- Managing tags in the console
- Managing tags in the AWS CLI
- Control access to X-Ray resources based on tags

Tag restrictions

The following restrictions apply to tags.

- Maximum number of tags per resource – 50
- Maximum key length – 128 Unicode characters
- Maximum value length – 256 Unicode characters
- Valid values for key and value – a-z, A-Z, 0-9, space, and the following characters: _ . : / = + - and @
- Tag keys and values are case sensitive.
- Don't use `aws:` as a prefix for keys; it's reserved for AWS use.

> **Note**
>
> You cannot edit or delete system tags.

Managing tags in the console

You can add optional tags as you create an X-Ray group or sampling rule. Tags can also be changed or deleted in the console later.

The following procedures explain how to add, edit, and delete tags for your groups and sampling rules in the X-Ray console.

Topics

- Add tags to a new group (console)
- Add tags to a new sampling rule (console)
- Edit or delete tags for a group (console)

- Edit or delete tags for a sampling rule (console)

Add tags to a new group (console)

As you create a new X-Ray group, you can add optional tags on the **Create group** page.

1. Sign in to the AWS Management Console and open the X-Ray console at https://console.aws.amazon.com/xray/home.
2. In the navigation pane, expand **Configuration**, and choose **Groups**.
3. Choose **Create group**.
4. On the **Create group** page, specify a name and filter expression for the group. For more information about these properties, see Configuring groups.
5. In **Tags**, enter a tag key, and optionally, a tag value. For example, you can enter a tag key of **`Stage`**, and a tag value of **`Production`**, to indicate that this group is for production use. As you add a tag, a new line appears for you to add another tag, if needed. See Tag restrictions in this topic for limitations on tags.
6. When you are finished adding tags, choose **Create group**.

Add tags to a new sampling rule (console)

As you create a new X-Ray sampling rule, you can add tags on the **Create sampling rule** page.

1. Sign in to the AWS Management Console and open the X-Ray console at https://console.aws.amazon.com/xray/home.
2. In the navigation pane, expand **Configuration**, and choose **Sampling**.
3. Choose **Create sampling rule**.
4. On the **Create sampling rule** page, specify a name, priority, limits, matching criteria, and matching attributes. For more information about these properties, see Configuring sampling rules.
5. In **Tags**, enter a tag key, and optionally, a tag value. For example, you can enter a tag key of **`Stage`**, and a tag value of **`Production`**, to indicate that this sampling rule is for production use. As you add a tag, a new line appears for you to add another tag, if needed. See Tag restrictions in this topic for limitations on tags.
6. When you are finished adding tags, choose **Create sampling rule**.

Edit or delete tags for a group (console)

You can change or delete tags on an X-Ray group on the **Edit group** page.

1. Sign in to the AWS Management Console and open the X-Ray console at https://console.aws.amazon.com/xray/home.
2. In the navigation pane, expand **Configuration**, and choose **Groups**.
3. In the **Groups** table, choose the name of a group.
4. On the **Edit group** page, in **Tags**, edit tag keys and values. You cannot have duplicate tag keys. Tag values are optional; you can delete values if desired. For more information about other properties on the **Edit group** page, see Configuring groups. See Tag restrictions in this topic for limitations on tags.
5. To delete a tag, choose **X** at the right of the tag.
6. When you are finished editing or deleting tags, choose **Update group**.

Edit or delete tags for a sampling rule (console)

You can change or delete tags on an X-Ray sampling rule on the **Edit sampling rule** page.

1. Sign in to the AWS Management Console and open the X-Ray console at https://console.aws.amazon.com/xray/home.
2. In the navigation pane, expand **Configuration**, and choose **Sampling**.
3. In the **Sampling rules** table, choose the name of a sampling rule.
4. In **Tags**, edit tag keys and values. You cannot have duplicate tag keys. Tag values are optional; you can delete values if desired. For more information about other properties on the **Edit sampling rule** page, see Configuring sampling rules. See Tag restrictions in this topic for limitations on tags.
5. To delete a tag, choose **X** at the right of the tag.
6. When you are finished editing or deleting tags, choose **Update sampling rule**.

Managing tags in the AWS CLI

You can add tags when you create an X-Ray group or sampling rule. You can also use the AWS CLI to create and manage tags. To update tags on an existing group or sampling rule, use the AWS X-Ray console, or the TagResource or UntagResource APIs.

Topics

- Add tags to a new X-Ray group or sampling rule (CLI)
- Add tags to an existing resource (CLI)
- List tags on a resource (CLI)
- Delete tags on a resource (CLI)

Add tags to a new X-Ray group or sampling rule (CLI)

To add optional tags as you're creating a new X-Ray group or sampling rule, use one of the following commands.

- To add tags to a new group, run the following command, replacing *group_name* with the name of your group, *mydomain.com* with the endpoint of your service, *key_name* with a tag key, and optionally, *value* with a tag value. For more information about how to create a group, see Groups.

  ```
  aws xray create-group \
     --group-name "group_name" \
     --filter-expression "service(\"mydomain.com\") {fault OR error}" \
     --tags [{"Key": "key_name","Value": "value"},{"Key": "key_name","Value": "value"}]
  ```

  The following is an example.

  ```
  aws xray create-group \
     --group-name "AdminGroup" \
     --filter-expression "service(\"mydomain.com\") {fault OR error}" \
     --tags [{"Key": "Stage","Value": "Prod"},{"Key": "Department","Value": "QA"}]
  ```

- To add tags to a new sampling rule, run the following command, replacing *key_name* with a tag key, and optionally, *value* with a tag value. This command specifies the values in the `--sampling-rule` parameter as a JSON file. For more information about how to create a sampling rule, see Sampling rules.

  ```
  aws xray create-sampling-rule \
     --cli-input-json file://file_name.json
  ```

The following are the contents of the JSON file *file_name.json* that is specified by the `--cli-input-json` parameter.

```
{
    "SamplingRule": {
        "RuleName": "rule_name",
        "RuleARN": "string",
        "ResourceARN": "string",
        "Priority": integer,
        "FixedRate": double,
        "ReservoirSize": integer,
        "ServiceName": "string",
        "ServiceType": "string",
        "Host": "string",
        "HTTPMethod": "string",
        "URLPath": "string",
        "Version": integer,
        "Attributes": {"attribute_name": "value","attribute_name": "value"...}
    }
    "Tags": [
           {
               "Key":"key_name",
               "Value":"value"
           },
           {
               "Key":"key_name",
               "Value":"value"
           }
          ]
}
```

The following command is an example.

```
aws xray create-sampling-rule \
   --cli-input-json file://9000-base-scorekeep.json
```

The following are the contents of the example `9000-base-scorekeep.json` file specified by the `--cli-input-json` parameter.

```
{
    "SamplingRule": {
        "RuleName": "base-scorekeep",
        "ResourceARN": "*",
        "Priority": 9000,
        "FixedRate": 0.1,
        "ReservoirSize": 5,
        "ServiceName": "Scorekeep",
        "ServiceType": "*",
        "Host": "*",
        "HTTPMethod": "*",
        "URLPath": "*",
        "Version": 1
    }
    "Tags": [
           {
               "Key":"Stage",
               "Value":"Prod"
           },
           {
               "Key":"Department",
               "Value":"QA"
           }
          ]
}
```

Add tags to an existing resource (CLI)

You can run the `tag-resource` command to add tags to an existing X-Ray group or sampling rule This method might be simpler than adding tags by running `update-group` or `update-sampling-rule`.

To add tags to a group or a sampling rule, run the following command, replacing the ARN with the ARN of the resource, and specifying the keys and optional values of tags that you want to add.

```
aws xray tag-resource \
   --resource-arn "ARN" \
   --tag-keys [{"Key":"key_name","Value":"value"}, {"Key":"key_name","Value":"value"}]
```

The following is an example.

```
aws xray tag-resource \
   --resource-arn "arn:aws:xray:us-east-2:01234567890:group/AdminGroup" \
   --tag-keys [{"Key": "Stage","Value": "Prod"},{"Key": "Department","Value": "QA"}]
```

List tags on a resource (CLI)

You can run the `list-tags-for-resource` command to list tags of an X-Ray group or sampling rule.

To list the tags that are associated with a group or a sampling rule, run the following command, replacing the ARN with the ARN of the resource.

```
aws xray list-tags-for-resource \
   --resource-arn "ARN"
```

The following is an example.

```
aws xray list-tags-for-resource \
   --resource-arn "arn:aws:xray:us-east-2:01234567890:group/AdminGroup"
```

Delete tags on a resource (CLI)

You can run the `untag-resource` command to remove tags from an X-Ray group or sampling rule.

To remove tags from a group or a sampling rule, run the following command, replacing the ARN with the ARN of the resource, and specifying the keys of tags that you want to remove.

You can remove only entire tags with the `untag-resource` command. To remove tag values, use the X-Ray console, or delete tags and add new tags with the same keys, but different or empty values.

```
aws xray untag-resource \
   --resource-arn "ARN" \
   --tag-keys ["key_name","key_name"]
```

The following is an example.

```
aws xray untag-resource \
   --resource-arn "arn:aws:xray:us-east-2:01234567890:group/group_name" \
   --tag-keys ["Stage","Department"]
```

Control access to X-Ray resources based on tags

You can attach tags to X-Ray groups or sampling rules, or pass tags in a request to X-Ray. To control access based on tags, you provide tag information in the condition element of a policy using the `xray:ResourceTag/`*`key-name`*, `aws:RequestTag/`*`key-name`*, or `aws:TagKeys` condition keys. To learn more about these condition keys, see Controlling access to AWS resources using resource tags.

To view an example identity-based policy for limiting access to a resource based on the tags on that resource, see Managing access to X-Ray groups and sampling rules based on tags.

Control access to X-Ray resources based on tags 634

Troubleshooting AWS X-Ray

This topic lists common errors and issues that you might encounter when using the X-Ray API, console, or SDKs. If you find an issue that is not listed here, you can use the **Feedback** button on this page to report it.

Sections

- X-Ray trace map and trace details pages
- X-Ray SDK for Java
- X-Ray SDK for Node.js
- The X-Ray daemon

X-Ray trace map and trace details pages

The following sections can help if you're having issues using the X-Ray trace map and Trace details page:

I don't see all of my CloudWatch logs

How to configure logs so that they appear in the X-Ray trace map and trace details pages depends on the service.

- API Gateway logs appear if logging is turned on in API Gateway.

Not all service map nodes support viewing the associated logs. View logs for the following node types:

- Lambda context
- Lambda function
- API Gateway stage
- Amazon ECS cluster
- Amazon ECS instance
- Amazon ECS service
- Amazon ECS task

- Amazon EKS cluster
- Amazon EKS namespace
- Amazon EKS node
- Amazon EKS pod
- Amazon EKS service

I don't see all of my alarms on the X-Ray trace map

The X-Ray trace map shows only the alert icon for a node if any alarms that are associated with that node are in the ALARM state.

The trace map associates alarms with nodes using the following logic:

- If the node represents an AWS service, then all alarms with the namespace associated with that service are associated with the node. For example, a node of type `AWS::Kinesis` is linked with all alarms that are based on metrics in the CloudWatch namespace `AWS/Kinesis`.
- If the node represents an AWS resource, then the alarms on that specific resource are linked. For example, a node of type `AWS::DynamoDB::Table` with the name “MyTable” is linked to all alarms that are based on a metric with the namespace `AWS/DynamoDB` and have the `TableName` dimension set to `MyTable`.
- If the node is of unknown type, which is identified by a dashed border around the name, then no alarms are associated with that node.

I don't see some AWS resources on the trace map

Not every AWS resource is represented by a dedicated node. Some AWS services are represented by a single node for all requests to the service. The following resource types are displayed with a node per resource:

- `AWS::DynamoDB::Table`
- `AWS::Lambda::Function`

  Lambda functions are represented by two nodes—one for the Lambda container, and one for the function. This helps to identify cold start problems with Lambda functions. Lambda container nodes are associated with alarms and dashboards in the same way as Lambda function nodes.
- `AWS::ApiGateway::Stage`

- `AWS::SQS::Queue`
- `AWS::SNS::Topic`

There are too many nodes on the trace map

Use X-Ray groups to break your map into multiple maps. For more information, see Using Filter Expressions with Groups.

X-Ray SDK for Java

Error: *Exception in thread "Thread-1" com.amazonaws.xray.exceptions.SegmentNotFoundException: Failed to begin subsegment named 'AmazonSNS': segment cannot be found.*

This error indicates that the X-Ray SDK attempted to record an outgoing call to AWS, but couldn't find an open segment. This can occur in the following situations:

- **A servlet filter is not configured** – The X-Ray SDK creates segments for incoming requests with a filter named `AWSXRayServletFilter`. Configure a servlet filter to instrument incoming requests.
- **You're using instrumented clients outside of servlet code** – If you use an instrumented client to make calls in startup code or other code that doesn't run in response to an incoming request, you must create a segment manually. See Instrumenting startup code for examples.
- **You're using instrumented clients in worker threads** – When you create a new thread, the X-Ray recorder loses its reference to the open segment. You can use the `getTraceEntity` and `setTraceEntity` methods to get a reference to the current segment or subsegment (`Entity`), and pass it back to the recorder inside of the thread. See Using instrumented clients in worker threads for an example.

X-Ray SDK for Node.js

Issue: *CLS does not work with Sequelize*

Pass the X-Ray SDK for Node.js namespace to Sequelize with the `cls` method.

```
var AWSXRay = require('aws-xray-sdk');
const Sequelize = require('sequelize');
Sequelize.cls = AWSXRay.getNamespace();
```

```
const sequelize = new Sequelize(...);
```

Issue: *CLS does not work with Bluebird*

Use `cls-bluebird` to get Bluebird working with CLS.

```
var AWSXRay = require('aws-xray-sdk');
var Promise = require('bluebird');
var clsBluebird = require('cls-bluebird');
clsBluebird(AWSXRay.getNamespace());
```

The X-Ray daemon

Issue: *The daemon is using the wrong credentials*

The daemon uses the AWS SDK to load credentials. If you use multiple methods of providing credentials, the method with the highest precedence is used. See Running the daemon for more information.

Document History for AWS X-Ray

The following table describes the important changes to the documentation for AWS X-Ray. For notification about updates to this documentation, you can subscribe to an RSS feed.

Latest documentation update: March 07, 2024

Change	Description	Date
Added end-of-support notice for AWS X-Ray SDKs and daemon	On February 25th, 2027, AWS X-Ray will discontinue support for AWS X-Ray SDKs and daemon. We recommend to migrate to OpenTelemetry. For more information, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation.	August 22, 2025
Added functionality	Migrating from X-Ray to OpenTelemetry. For more information, see Migrating from X-Ray instrumentation to OpenTelemetry instrumentation.	June 13, 2025
Added functionality	AWS X-Ray now supports Transaction Search. For more information, see Transaction Search.	November 21, 2024
Added functionality	AWS X-Ray now supports OpenTelemetry Protocol (OTLP) Endpoint. For more information, see OpenTelemetry.	November 21, 2024

Added functionality	X-Ray now logs data events, including `PutTraceSegments` , `GetTraceSummaries` , and `BatchGetTraces` to AWS CloudTrail. X-Ray also now logs the `GetSamplingStatisticSummaries` management event to CloudTrail. For more information, see Logging X-Ray API calls with AWS CloudTrail.	March 7, 2024
Added functionality	X-Ray now supports trace IDs created via OpenTelemetry or any other framework which conforms to the W3C Trace Context specification. For more information, see Sending trace data to X-Ray.	October 25, 2023
Added functionality	You can now configure Amazon SNS active tracing, enabling you to trace and analyze requests as they travel through your Amazon SNS topics. For more information, see Amazon SNS and AWS X-Ray.	February 8, 2023
Updated X-Ray SDK for Node.js topic	Added details for instrumenting clients using the AWS SDK for JavaScript V3. For details, see Tracing AWS SDK calls with the X-Ray SDK for Node.js.	February 7, 2023

Updated IAM managed policy details	Added IAM permission for cross-account observability to the `AWSXRayReadOnlyAccess` , `AWSXRayFullAccess` and `AWSXrayCrossAccountSharingConfiguration` managed policies. For details, see IAM managed policies for X-Ray.	February 7, 2023
Added functionality	AWS X-Ray now supports *cross-account observability*, enabling you to monitor and troubleshoot applications that span across multiple accounts within an AWS Region. For details, see Cross-account tracing.	November 27, 2022
Added functionality	You can now view linked traces between message producers, an Amazon SQS queue, and consumers, providing a connected view of traces sent from event-driven applications. For more information, see tracing event-driven applications.	November 20, 2022
Updated IAM managed policy details	Added IAM permission for listing resource policies to the `AWSXRayReadOnlyAccess` managed policy. For details, see IAM managed policies for X-Ray.	November 15, 2022

Updated IAM console permissions and managed policy details	The set of IAM permissions the X-Ray console uses has been updated, along with the description of the `AWSXRayReadOnlyAccess` managed policy. For details, see Using the X-Ray console.	November 11, 2022
Added AWS Distro for OpenTelemetry Ruby	AWS Distro for OpenTelemetry (ADOT) provides a single set of open source APIs, libraries, and agents to collect distributed traces and metrics. ADOT Ruby enables you to instrument your Ruby application for X-Ray and other tracing back-ends. For more information, see AWS Distro for OpenTelemetry Ruby.	February 7, 2022
Added functionality	You can now view traces and configure X-Ray from the CloudWatch console. For more information, see X-Ray console.	January 24, 2022
Integrated CloudWatch RUM	With AWS X-Ray and CloudWatch RUM, you can analyze and debug the request path starting from end users of your application through downstream AWS managed services. For more information, see CloudWatch RUM and AWS X-Ray.	December 3, 2021

Integrated AWS Distro for OpenTelemetry	The AWS Distro for OpenTelemetry (ADOT) provides a single set of open source APIs, libraries, and agents to collect distributed traces and metrics. ADOT enables you to instrument your application for X-Ray and other tracing back-ends . For more information, see Instrumenting your app.	September 23, 2021
Added functionality	AWS X-Ray now integrates with Amazon Virtual Private Cloud, enabling resources in your Amazon VPC to communicate with the X-Ray service without going through the public internet. For more information, see Using AWS X-Ray with VPC endpoints.	May 20, 2021
Added functionality	AWS X-Ray now integrates with AWS CloudFormation, enabling you to provision and configure X-Ray resources . For more information, see Creating X-Ray resources with CloudFormation.	May 6, 2021

Added functionality	AWS X-Ray now integrates with Amazon EventBridge to trace events that are passed through EventBridge. This provides users with a more complete view of their system. For more information, see Amazon EventBridge and AWS X-Ray.	March 2, 2021
Added daemon to ECR	The daemon can now be downloaded from Amazon ECR. For more information, see Downloading the daemon.	March 1, 2021
Added functionality	AWS X-Ray now supports insights related notifications to Amazon EventBridge. This allows you to take automatic actions on insights using EventBridge. For more information, see Insights Notifications.	October 15, 2020
Added Downloadable Daemons	AWS X-Ray introduces support daemon for Linux ARM64. For more information, see AWS X-Ray daemonbrazil ws	October 1, 2020

Added functionality	AWS X-Ray now supports active integration with Amazon CloudWatch Synthetics. This allows you to see details about a Synthetics canary client node such as response time and status. You can also do analysis in the Analytics console based on information from a Synthetics canary client node. For more information, see Debugging CloudWatch synthetics canaries using X-Ray .	September 24, 2020
Added functionality	AWS X-Ray now supports tracing end-to-end workflows for AWS Step Functions . You can visualize the components of your state machine, identify performance bottlenecks, and troubleshoot requests that resulted in an error. For more information, see AWS Step Functions and AWS X-Ray.	September 14, 2020

Added functionality	AWS X-Ray introduces insights to continuously analyze trace data in your account to identify emergent issues in your applications. Insights records incidents and track incident impact until resolution. For more information, see Using insights in the AWS X-Ray console	September 3, 2020
Added functionality	AWS X-Ray introduces the Java auto-instrumentation agent, enabling customers to collect trace data without having to modify existing Java-based application. You can now trace Java web and servlet based applications with minimal configuration change and no code change. For more information, see AWS X-Ray auto-instrumentation agent for Java.	September 3, 2020
Added functionality	AWS X-Ray has added a new **Groups** page to the X-Ray console to help ease the creation and management of groups of traces. For more information, see Configuring groups in the X-Ray console.	August 24, 2020

Added functionality	AWS X-Ray now lets you add tags to groups and sampling rules. You can also control access to groups and sampling rules based on tags. For more information, see Tagging X-Ray sampling rules and groups and Managing access to X-Ray groups and sampling rules based on tags.	August 24, 2020