Serverless Architectures with AWS Lambda

AWS Whitepaper

Serverless Architectures with AWS Lambda: AWS Whitepaper

Table of Contents

Serverless Architectures with AWS Lambda

Publication date: **November 2017** (*Document Revisions* (p. 33))

Abstract

Since its introduction at AWS re:Invent in 2014, AWS Lambda has continued to be one of the fastest growing AWS services. With its arrival, a new application architecture paradigm was created—referred to as **serverless**. AWS now provides a number of different services that allow you to build full application stacks without the need to manage any servers. Use cases like web or mobile backends, real-time data processing, chatbots and virtual assistants, Internet of Things (IoT) backends, and more can all be fully serverless. For the logic layer of a serverless application, you can execute your business logic using AWS Lambda. Developers and organizations are finding that AWS Lambda is enabling much faster development speed and experimentation than is possible when deploying applications in a traditional server-based environment.

This whitepaper is meant to provide you with a broad overview of AWS Lambda, its features, and a slew of recommendations and best practices for building your own serverless applications on AWS.

Introduction - What Is Serverless?

Serverless most often refers to serverless applications. Serverless applications are ones that don't require you to provision or manage any servers. You can focus on your core product and business logic instead of responsibilities like operating system (OS) access control, OS patching, provisioning, right-sizing, scaling, and availability. By building your application on a serverless platform, the platform manages these responsibilities for you.

For service or platform to be considered serverless, it should provide the following capabilities:

- **No server management** – You don't have to provision or maintain any servers. There is no software or runtime to install, maintain, or administer.
- **Flexible scaling** – You can scale your application automatically or by adjusting its capacity through toggling the units of consumption (for example, throughput, memory) rather than units of individual servers.
- **High availability** – Serverless applications have built-in availability and fault tolerance. You don't need to architect for these capabilities because the services running the application provide them by default.
- **No idle capacity** – You don't have to pay for idle capacity. There is no need to pre-provision or over-provision capacity for things like compute and storage. There is no charge when your code isn't running.

The AWS Cloud provides many different services that can be components of a serverless application. These include capabilities for:

- Compute – AWS Lambda
- APIs – Amazon API Gateway
- Storage – Amazon Simple Storage Service (Amazon S3)
- Databases –Amazon DynamoDB
- Interprocess messaging – Amazon Simple Notification Service (Amazon SNS) and Amazon Simple Queue Service (Amazon SQS)
- Orchestration – AWS Step Functions and Amazon CloudWatch Events
- Analytics – Amazon Kinesis

This whitepaper will focus on AWS Lambda, the compute layer of your serverless application where your code is executed, and the AWS developer tools and services that enable best practices when building and maintaining serverless applications with Lambda.

AWS Lambda—the Basics

Lambda is a high-scale, provision-free serverless compute offering based on **functions**. It provides the cloud logic layer for your application. Lambda functions can be triggered by a variety of events that occur on AWS or on supporting third-party services. They enable you to build reactive, event-driven systems. When there are multiple, simultaneous events to respond to, Lambda simply runs more copies of the function in parallel. Lambda functions scale precisely with the size of the workload, down to the individual request. Thus, the likelihood of having an idle server or container is extremely low. Architectures that use Lambda functions are designed to reduce wasted capacity.

Lambda can be described as a type of serverless Function-as-a-Service (FaaS). FaaS is one approach to building event-driven computing systems. It relies on functions as the unit of deployment and execution. Serverless FaaS is a type of FaaS where no virtual machines or containers are present in the programming model and where the vendor provides provision-free scalability and built-in reliability.

Figure 1 shows the relationship among event-driven computing, FaaS, and serverless FaaS.

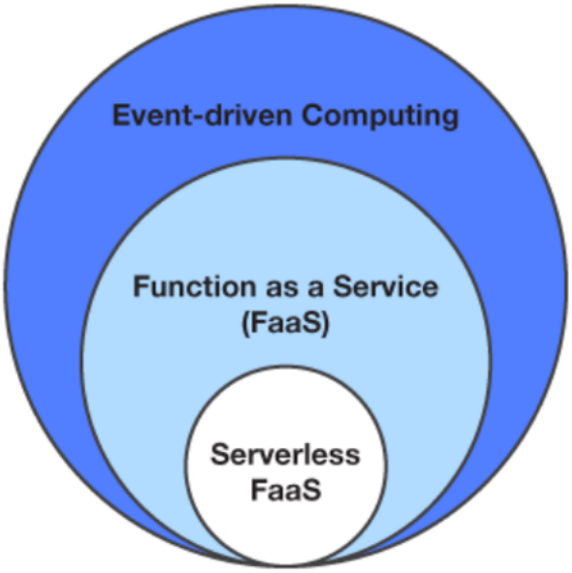

Figure 1: The relationship among event-driven computing, FaaS, and serverless FaaS

With Lambda, you can run code for virtually any type of application or backend service. Lambda runs and scales your code with high availability.

Each Lambda function you create contains the **code** you want to execute, the **configuration** that defines how your code is executed and, optionally, one or more **event sources** that detect events and invoke your function as they occur. These elements are covered in more detail in the next section (p. 5).

An example event source is API Gateway, which can invoke a Lambda function anytime an API method created with API Gateway receives an HTTPS request. Another example is Amazon SNS, which has the ability to invoke a Lambda function anytime a new message is posted to an SNS topic. Many event source options can trigger your Lambda function. For the full list, see this documentation. Lambda also provides a RESTful service API, which includes the ability to directly invoke a Lambda function. You can use this API to execute your code directly without configuring another event source.

You don't need to write any code to integrate an event source with your Lambda function, manage any of the infrastructure that detects events and delivers them to your function, or manage scaling your

Lambda function to match the number of events that are delivered. You can focus on your application logic and configure the event sources that cause your logic to run.

Your Lambda function runs within a (simplified) architecture that looks like the one shown in Figure 2.

Figure 2: Simplified architecture of a running Lambda function

Once you configure an event source for your function, your code is invoked when the event occurs. Your code can execute any business logic, reach out to external web services, integrate with other AWS services, or anything else your application requires. All of the same capabilities and software design principles that you're used to for your language of choice will apply when using Lambda. Also, because of the inherent decoupling that is enforced in serverless applications through integrating Lambda functions and event sources, it's a natural fit to build microservices using Lambda functions.

With a basic understanding of serverless principles and Lambda, you might be ready to start writing some code. The following resources will help you get started with Lambda immediately:

- Hello World tutorial: http://docs.aws.amazon.com/lambda/latest/dg/get-started-create-function.html
- Serverless workshops and walkthroughs for building sample applications: https://github.com/awslabs/aws-serverless-workshops

AWS Lambda—Diving Deeper

The remainder of this whitepaper will help you understand the components and features of Lambda, followed by best practices for various aspects of building and owning serverless applications using Lambda.

Let's begin our deep dive by further expanding and explaining each of the major components of Lambda that we described in the introduction: function code, event sources, and function configuration.

Lambda Function Code

At its core, you use Lambda to execute code. This can be code that you've written in any of the languages supported by Lambda (Java, Node.js, Python, or C# as of this publication), as well as any code or packages you've uploaded alongside the code that you've written. You're free to bring any libraries, artifacts, or compiled native binaries that can execute on top of the runtime environment as part of your function code package. If you want, you can even execute code you've written in another programming language (PHP, Go, SmallTalk, Ruby, etc.), as long as you stage and invoke that code from within one of the support languages in the AWS Lambda runtime environment (see this tutorial).

The Lambda runtime environment is based on an Amazon Linux AMI (see current environment details here), so you should compile and test the components you plan to run inside of Lambda within a matching environment. To help you perform this type of testing prior to running within Lambda, AWS provides a set of tools called AWS SAM Local to enable local testing of Lambda functions. We discuss these tools in the Serverless Development Best Practices section (p. 24) of this whitepaper.

The Function Code Package

The function code **package** contains all of the assets you want to have available locally upon execution of your code. A package will, at minimum, include the code function you want the Lambda service to execute when your function is invoked. However, it might also contain other assets that your code will reference upon execution, for example, additional files, classes, and libraries that your code will import, binaries that you would like to execute, or configuration files that your code might reference upon invocation. The maximum size of a function code package is 50 MB compressed and 250MB extracted at the time of this publication. (For the full list of AWS Lambda limits, see this documentation.)

When you create a Lambda function (through the AWS Management Console, or using the CreateFunction API) you can reference the S3 bucket and object key where you've uploaded the package. Alternatively, you can upload the code package directly when you create the function. Lambda will then store your code package in an S3 bucket managed by the service. The same options are available when you publish updated code to existing Lambda functions (through the UpdateFunctionCode API).

As events occur, your code package will be downloaded from the S3 bucket, installed in the Lambda runtime environment, and invoked as needed. This happens on demand, at the scale required by the number of events triggering your function, within an environment managed by Lambda.

The Handler

When a Lambda function is invoked, code execution begins at what is called the **handler**. The handler is a specific code method (Java, C#) or function (Node.js, Python) that you've created and included in your package. You specify the handler when creating a Lambda function. Each language supported by Lambda has its own requirements for how a function handler can be defined and referenced within the package.

The following links will help you get started with each of the supported languages.

Language	Example Handler Definition
Java	`MyOutput output handlerName(MyEvent event, Context context) { ... }`
Node.js	`exports.handlerName = function(event, context, callback) { ... // callback parameter is optional }`
Python	`def handler_name(event, context): ... return some_value`
C#	`myOutput HandlerName(MyEvent event, ILambdaContext context) { ... }`

Once the handler is successfully invoked inside your Lambda function, the runtime environment belongs to the code you've written. Your Lambda function is free to execute any logic you see fit, driven by the code you've written that starts in the handler. This means your handler can call other methods and functions within the files and classes you've uploaded. Your code can import third-party libraries that you've uploaded, and install and execute native binaries that you've uploaded (as long as they can run on Amazon Linux). It can also interact with other AWS services or make API requests to web services that it depends on, etc.

The Event Object

When your Lambda function is invoked in one of the supported languages, one of the parameters provided to your handler function is an **event object**. The event differs in structure and contents, depending on which event source created it. The contents of the event parameter include all of the data and metadata your Lambda function needs to drive its logic. For example, an event created by API Gateway will contain details related to the HTTPS request that was made by the API client (for example, path, query string, request body), whereas an event created by Amazon S3 when a new object is created will include details about the bucket and the new object.

The Context Object

Your Lambda function is also provided with a **context object**. The context object allows your function code to interact with the Lambda execution environment. The contents and structure of the context object vary, based on the language runtime your Lambda function is using, but at minimum it will contain:

- **AWS RequestId** – Used to track specific invocations of a Lambda function (important for error reporting or when contacting AWS Support).
- **Remaining time** – The amount of time in milliseconds that remain before your function timeout occurs (Lambda functions can run a maximum of 300 seconds as of this publishing, but you can configure a shorter timeout).

- **Logging** – Each language runtime provides the ability to stream log statements to Amazon CloudWatch Logs. The context object contains information about which CloudWatch Logs stream your log statements will be sent to. For more information about how logging is handled in each language runtime, see the following:

- Java
- Node.js
- Python
- C#

Writing Code for AWS Lambda—Statelessness and Reuse

It's important to understand the central tenant when writing code for Lambda: **your code cannot make assumptions about state**. This is because Lambda fully manages when a new function container will be created and invoked for the first time. A container could be getting invoked for the first time for a number of reasons. For example, the events triggering your Lambda function are increasing in concurrency beyond the number of containers previously created for your function, an event is triggering your Lambda function for the first time in several minutes, etc. While Lambda is responsible for scaling your function containers up and down to meet actual demand, your code needs to be able to operate accordingly. Although Lambda won't interrupt the processing of a specific invocation that's already in flight, your code doesn't need to account for that level of volatility.

This means that your code cannot make any assumptions that state will be preserved from one invocation to the next. However, each time a function container is created and invoked, it remains active and available for subsequent invocations for at least a few minutes before it is terminated. When subsequent invocations occur on a container that has already been active and invoked at least once before, we say that invocation is running on a **warm container**. When an invocation occurs for a Lambda function that requires your function code package to be created and invoked for the first time, we say the invocation is experiencing a **cold start**.

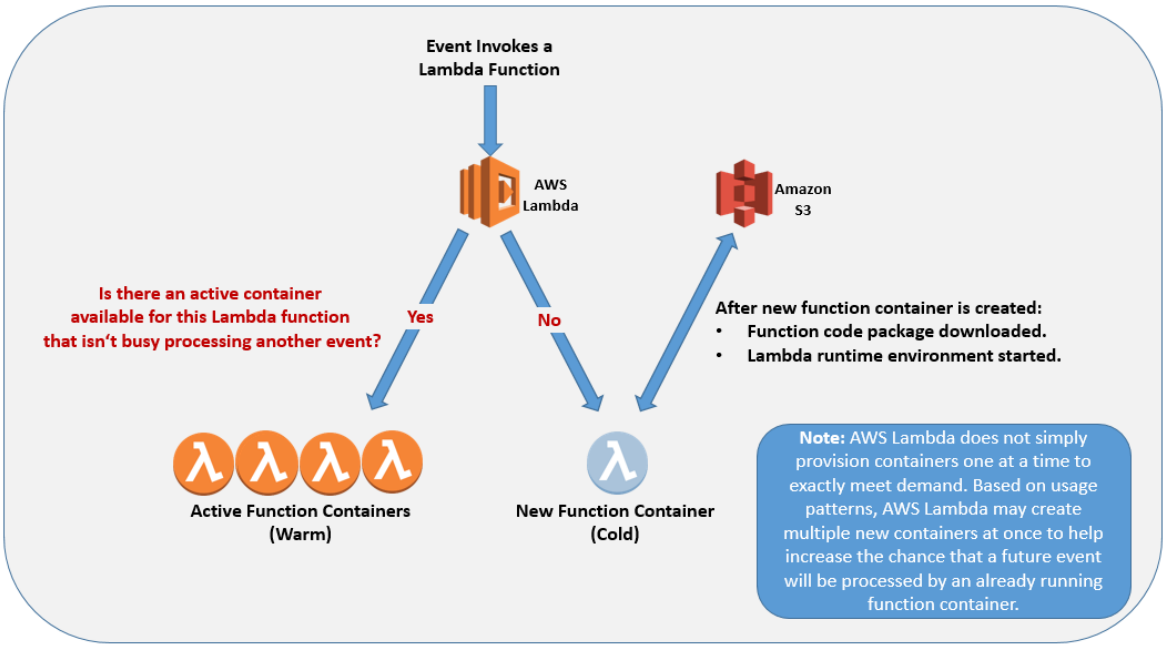

Figure 3: Invocations of warm function containers and cold function containers

Depending on the logic your code is executing, understanding how your code can take advantage of a warm container can result in faster code execution inside of Lambda. This, in turn, results in quicker responses and lower cost. For more details and examples of how to improve your Lambda function performance by taking advantage of warm containers, see the Best Practices section (p. 26) later in this whitepaper.

Overall, each language that Lambda supports has its own model for packaging source code and possibilities for optimizing it. Visit this page to get started with each of the supported languages.

Lambda Function Event Sources

Now that you know what goes into the code of a Lambda function, let's look at the **event sources**, or triggers, that invoke your code. While Lambda provides the Invoke API that enables you to directly invoke your function, you will likely only use it for testing and operational purposes. Instead, you can associate your Lambda function with event sources occurring within AWS services that will invoke your function as needed. You don't have to write, scale, or maintain any of the software that integrates the event source with your Lambda function.

Invocation Patterns

There are two models for invoking a Lambda function:

- **Push Model** – Your Lambda function is invoked every time a particular event occurs within another AWS service (for example, a new object is added to an S3 bucket).
- **Pull Model** – Lambda polls a data source and invokes your function with any new records that arrive at the data source, batching new records together in a single function invocation (for example, new records in an Amazon Kinesis or Amazon DynamoDB stream).

  Also, a Lambda function can be executed synchronously or asynchronously. You choose this using the parameter **InvocationType** that's provided when invoking a Lambda function. This parameter has three possible values:

- **RequestResponse** – Execute synchronously.
- **Event** – Execute asynchronously.
- **DryRun** – Test that the invocation is permitted for the caller, but don't execute the function.

Each event source dictates how your function can be invoked. The event source is also responsible for crafting its own event parameter, as we discussed earlier.

The following tables provide details about how some of the more popular event sources can integrate with your Lambda functions. You can find the full list of supported event sources here.

Push Model Event Sources

Amazon S3

Invocation Model	Push
Invocation Type	Event

Invocation Model	Push
Description	S3 event notifications (such as ObjectCreated and ObjectRemoved) can be configured to invoke a Lambda function as they are published.
Example Use Cases	Create image modifications (thumbnails, different resolutions, watermarks, etc.) for images that users upload to an S3 bucket through your application. Process raw data uploaded to an S3 bucket and move transformed data to another S3 bucket as part of a big data pipeline.

Amazon API Gateway

Invocation Model	Push
Invocation Type	Event or RequestResponse
Description	The API methods you create with API Gateway can use a Lambda function as their service backend. If you choose Lambda as the integration type for an API method, your Lambda function is invoked synchronously (the response of your Lambda function serves as the API response). With this integration type, API Gateway can also act as a simple proxy to a Lambda function. API Gateway will perform no processing or transformation on its own and will pass along all the contents of the request to Lambda. If you want an API to invoke your function asynchronously as an event and return immediately with an empty response, you can use API Gateway as an AWS Service Proxy and integrate with the Lambda Invoke API, providing the Event InvocationType in the request header. This is a great option if your API clients don't need any information back from the request and you want the fastest response time possible. (This option is great for pushing user interactions on a website or app to a service backend for analysis.)
Example Use Cases	Web service backends (web application, mobile app, microservice architectures, etc.) Legacy service integration (a Lambda function to transform a legacy SOAP backend into a new modern REST API). Any other use cases where HTTPS is the appropriate integration mechanism between application components.

Amazon SNS

Invocation Model	Push
Invocation Type	Event
Description	Messages that are published to an SNS topic can be delivered as events to a Lambda function.
Example Use Cases	Automated responses to CloudWatch alarms. Processing of events from other services (AWS or otherwise) that can natively publish to SNS topics.

AWS CloudFormation

Invocation Model	Push
Invocation Type	RequestResponse
Description	As part of deploying AWS CloudFormation stacks, you can specify a Lambda function as a custom resource to execute any custom commands and provide data back to the ongoing stack creation.
Example Use Cases	Extend AWS CloudFormation capabilities to include AWS service features not yet natively supported by AWS CloudFormation. Perform custom validation or reporting at key stages of the stack creation/update/delete process.

Amazon CloudWatch Events

Invocation Model	Push
Invocation Type	Event
Description	Many AWS services publish resource state changes to CloudWatch Events. Those events can then be filtered and routed to a Lambda function for automated responses.
Example Use Cases	Event-driven operations automation (for example, take action each time a new EC2 instance is launched, notify an appropriate mailing list when AWS Trusted Advisor reports a new status change). Replacement for tasks previously accomplished with cron (CloudWatch Events supports scheduled events).

Amazon Alexa

Invocation Model	Push
Invocation Type	RequestResponse
Description	You can write Lambda functions that act as the service backend for Amazon Alexa Skills. When an Alexa user interacts with your skill, Alexa's Natural Language Understand and Processing capabilities will deliver their interactions to your Lambda functions.
Example Use Cases	An Alexa skill of your own.

Pull Model Event Sources

Amazon DynamoDB

Invocation Model	Pull
Invocation Type	Request/Response
Description	Lambda will poll a DynamoDB stream multiple times per second and invoke your Lambda function with the batch of updates that have been published to the stream since the last batch. You can configure the batch size of each invocation.
Example Use Cases	Application-centric workflows that should be triggered as changes occur in a DynamoDB table (for example, a new user registered, an order was placed, a friend request was accepted, etc.). Replication of a DynamoDB table to another region (for disaster recovery) or another service (shipping as logs to an S3 bucket for backup or analysis).

Amazon Kinesis Streams

Invocation Model	Pull
Invocation Type	Request/Response
Description	Lambda will poll a Kinesis stream, once per second for each stream shard, and invoke your Lambda function with the next records in the shard. You can define the batch size for the number of records delivered to your function at a time, as well as the number of Lambda function containers executing concurrently (number of stream shards = number of concurrent function containers).
Example Use Cases	Real-time data processing for big data pipelines.

Invocation Model	Pull
	Real-time alerting/monitoring of streaming log statements or other application events.

Lambda Function Configuration

After you write and package your Lambda function code, on top of choosing which event sources will trigger your function, you have various configuration options to set that define how your code is executed within Lambda.

Function Memory

To define the resources allocated to your executing Lambda function, you're provided with a single dial to increase/decrease function resources: memory/RAM. You can allocate 128 MB of RAM up to 1.5 GB of RAM to your Lambda function. Not only will this dictate the amount of memory available to your function code during execution, but the same dial will also influence the CPU and network resources available to your function.

Selecting the appropriate memory allocation is a very important step when optimizing the price and performance of any Lambda function. Please review the best practices later in this whitepaper for more specifics on optimizing performance.

Versions and Aliases

There are times where you might need to reference or revert your Lambda function back to code that was previously deployed. Lambda lets you **version** your AWS Lambda functions. Each and every Lambda function has a default version built in: $LATEST. You can address the most recent code that has been uploaded to your Lambda function through the $LATEST version. You can take a snapshot of the code that's currently referred to by $LATEST and create a numbered version through the PublishVersion API. Also, when updating your function code through the UpdateFunctionCode API, there is an optional Boolean parameter, `publish`. By setting `publish: true` in your request, Lambda will create a new Lambda function version, incremented from the last published version.

You can invoke each version of your Lambda function independently, at any time. Each version has its own Amazon Resource Name (ARN), referenced like this:

```
arn:aws:lambda:[region]:[account]:function:[fn_name]:[version]
```

When calling the Invoke API or creating an event source for your Lambda function, you can also specify a specific version of the Lambda function to be executed. If you don't provide a version number, or use the ARN that doesn't contain the version number, $LATEST is invoked by default.

It's important to know that a Lambda function container is specific to a particular version of your function. So, for example, if there are already several function containers deployed and available in the Lambda runtime environment for version 5 of the function, version 6 of the same function will not be able to execute on top of the existing version 5 containers—a different set of containers will be installed and managed for each function version.

Invoking your Lambda functions by their version numbers can be useful during testing and operational activities. However, we don't recommend having your Lambda function be triggered by a specific version number for real application traffic. Doing so would require you to update all of the triggers and clients

invoking your Lambda function to point at a new function version each time you wanted to update your code. Lambda **aliases** should be used here, instead. A function alias allows you to invoke and point event sources to a specific Lambda function version.

However, you can update what version that alias refers to at any time. For example, your event sources and clients that are invoking version number 5 through the alias `live` may cut over to version number 6 of your function as soon as you update the `live` alias to instead point at version number 6. Each alias can be referred to within the ARN, similar to when referring to a function version number:

```
arn:aws:lambda:[region]:[account]:function:[fn_name]:[alias]
```

> **Note**
> An alias is simply a pointer to a specific version number. This means that if you have multiple different aliases pointed to the same Lambda function version at once, requests to each alias are executed on top of the same set of installed function containers. This is important to understand so that you don't mistakenly point multiple aliases at the same function version number, if requests for each alias are intended to be processed separately.

Here are some example suggestions for Lambda aliases and how you might use them:

- **live/prod/active** – This could represent the Lambda function version that your production triggers or that clients are integrating with.
- **blue/green** – Enable the blue/green deployment pattern through use of aliases.
- **debug** – If you've created a testing stack to debug your applications, it can integrate with an alias like this when you need to perform a deeper analysis.

Creating a good, documented strategy for your use of function aliases enables you to have sophisticated serverless deployment and operations practices.

IAM Role

AWS Identity and Access Management (IAM) provides the capability to create IAM policies that define permissions for interacting with AWS services and APIs. Policies can be associated with **IAM roles**. Any access key ID and secret access key generated for a particular role is authorized to perform the actions defined in the policies attached to that role. For more information about IAM best practices, see this documentation.

In the context of Lambda, you assign an IAM role (called an **execution role**) to each of your Lambda functions. The IAM policies attached to that role define what AWS service APIs your function code is authorized to interact with. There are two benefits:

- Your source code isn't required to perform any AWS credential management or rotation to interact with the AWS APIs. Simply using the AWS SDKs and the default credential provider results in your Lambda function automatically using temporary credentials associated with the execution role assigned to the function.
- Your source code is decoupled from its own security posture. If a developer attempts to change your Lambda function code to integrate with a service that the function doesn't have access to, that integration will fail due to the IAM role assigned to your function. (Unless they have used IAM credentials that are separate from the execution role, you should use static code analysis tools to ensure that no AWS credentials are present in your source code).

It's important to assign each of your Lambda functions a specific, separate, and least-privilege IAM role. This strategy ensures that each Lambda function can evolve independently without increasing the authorization scope of any other Lambda functions.

Lambda Function Permissions

You can define which push model event sources are allowed to invoke a Lambda function through a concept called **permissions**. With permissions, you declare a **function policy** that lists the AWS Resource Names (ARNs) that are allowed to invoke a function.

For pull model event sources (for example, Kinesis streams and DynamoDB streams), you need to ensure that the appropriate actions are permitted by the IAM execution role assigned to your Lambda function. AWS provides a set of managed IAM roles associated with each of the pull-based event sources if you don't want to manage the permissions required. However, to ensure least privilege IAM policies, you should create your own IAM roles with resource-specific policies to permit access to just the intended event source.

Network Configuration

Executing your Lambda function occurs through the use of the **Invoke** API that is part of the AWS Lambda service APIs; so, there is no direct inbound network access to your function to manage. However, your function code might need to integrate with external dependencies (internal or publically hosted web services, AWS services, databases, etc.). A Lambda function has two broad options for outbound network connectivity:

- **Default** – Your Lambda function communicates from inside a virtual private cloud (VPC) that is managed by Lambda. It can connect to the internet, but not to any privately deployed resources running within your own VPCs.
- **VPC** – Your Lambda function communicates through an **Elastic Network Interface (ENI)** that is provisioned within the VPC and subnets you choose within your own account. These ENIs can be assigned security groups, and traffic will route based on the route tables of the subnets those ENIs are placed within—just the same as if an EC2 instance were placed in the same subnet.

If your Lambda function doesn't require connectivity to any privately deployed resources, we recommend you select the default networking option. Choosing the VPC option will require you to manage:

- Selecting appropriate subnets to ensure multiple Availability Zones are being used for the purposes of high availability.
- Allocating the appropriate number of IP addresses to each subnet to manage capacity.
- Implementing a VPC network design that will permit your Lambda functions to have the connectivity and security required.
- An increase in Lambda cold start times if your Lambda function invocation patterns require a new ENI to be created just in time. (ENI creation can take many seconds today.)

However, if your use case requires private connectivity, use the VPC option with Lambda. For deeper guidance if you plan to deploy your Lambda functions within your own VPC, see this documentation.

Environment Variables

Software Development Life Cycle (SDLC) best practice dictates that developers separate their code and their config. You can achieve this by using environment variables with Lambda. Environment variables for Lambda functions enable you to dynamically pass data to your function code and libraries without making changes to your code. Environment variables are key-value pairs that you create and modify as part of your function configuration. By default, these variables are encrypted at rest. For any sensitive information that will be stored as a Lambda function environment variable, we recommend you encrypt those values using the AWS Key Management Service (AWS KMS) prior to function creation, storing the encrypted cyphertext as the variable value. Then have your Lambda function decrypt that variable in memory at execution time.

Here are some examples of how you might decide to use environment variables:

- Log settings (FATAL, ERROR, INFO, DEBUG, etc.)
- Dependency and/or database connection strings and credentials
- Feature flags and toggles

Each version of your Lambda function can have its own environment variable values. However, once the values are established for a numbered Lambda function version, they cannot be changed. To make changes to your Lambda function environment variables, you can change them to the $LATEST version and then publish a new version that contains the new environment variable values. This enables you to always keep track of which environment variable values are associated with a previous version of your function. This is often important during a rollback procedure or when triaging the past state of an application.

Dead Letter Queues

Even in the serverless world, exceptions can still occur. (For example, perhaps you've uploaded new function code that doesn't allow the Lambda event to be parsed successfully, or there is an operational event within AWS that is preventing the function from being invoked.) For asynchronous event sources (the event **InvocationType**), AWS owns the client software that is responsible for invoking your function. AWS does not have the ability to synchronously notify you if the invocations are successful or not as invocations occur. If an exception occurs when trying to invoke your function in these models, the invocation will be attempted two more times (with back-off between the retries). After the third attempt, the event is either discarded or placed onto a **dead letter queue**, if you configured one for the function.

A dead letter queue is either an SNS topic or SQS queue that you have designated as the destination for all failed invocation events. If a failure event occurs, the use of a dead letter queue allows you to retain just the messages that failed to be processed during the event. Once your function is able to be invoked again, you can target those failed events in the dead letter queue for reprocessing. The mechanisms for reprocessing/retrying the function invocation attempts placed on to your dead letter queue is up to you. For more information about dead letter queues, see this tutorial. You should use dead letter queues if it's important to your application that *all* invocations of your Lambda function complete eventually, even if execution is delayed.

Timeout

You can designate the maximum amount of time a single function execution is allowed to complete before a timeout is returned. The maximum timeout for a Lambda function is 300 seconds at the time of this publication, which means a single invocation of a Lambda function cannot execute longer than 300 seconds. You should not always set the timeout for a Lambda function to the maximum. There are many cases where an application should fail fast. Because your Lambda function is billed based on execution time in 100-ms increments, avoiding lengthy timeouts for functions can prevent you from being billed while a function is simply waiting to timeout (perhaps an external dependency is unavailable, you've accidentally programmed an infinite loop, or another similar scenario).

Also, once execution completes or a timeout occurs for your Lambda function and a response is returned, all execution ceases. This includes any background processes, subprocesses, or asynchronous processes that your Lambda function might have spawned during execution. So you should not rely on background or asynchronous processes for critical activities. Your code should ensure those activities are completed prior to timeout or returning a response from your function.

Serverless Best Practices

Now that we've covered the components of a Lambda-based serverless application, let's cover some recommended best practices. There are many SDLC and server-based architecture best practices that are also true for serverless architectures: eliminate single points of failure, test changes prior to deployment, encrypt sensitive data, etc.

However, achieving best practices for serverless architectures can be a different task because of how different the operating model is. You don't have access to, or concerns about, an operating system or any lower-level components in the infrastructure. Because of this, your focus is solely on your own application code/architecture, the development processes you follow, and the features of the AWS services your application leverages that enable you to follow best practices.

First, we review a set of best practices for designing your serverless architecture according to the AWS Well-Architected Framework. Then, we cover some best practices and recommendations for your development process when building serverless applications.

Serverless Architecture Best Practices

The AWS Well-Architected Framework includes strategies to help you compare your workload against our best practices, and obtain guidance to produce stable and efficient systems so you can focus on functional requirements. It is based on five pillars: security, reliability, performance efficiency, cost optimization, and operational excellence. Many of the guidelines in the framework apply to serverless applications. However, there are specific implementation steps or patterns that are unique to serverless architectures. In the following sections, we cover a set of recommendations that are serverless-specific for each of the Well-Architected pillars.

Security Best Practices

Designing and implementing security into your applications should always be priority number one—this doesn't change with a serverless architecture. The major difference for securing a serverless application compared to a server-hosted application is obvious—there is no server for you to secure. However, you still need to think about your application's security. There is still a shared responsibility model for serverless security.

With Lambda and serverless architectures, rather than implementing application security through things like antivirus/malware software, file integrity monitoring, intrusion detection/prevention systems, firewalls, etc., you ensure security best practices through writing secure application code, tight access control over source code changes, and following AWS security best practices for each of the services that your Lambda functions integrate with.

The following is a brief list of serverless security best practices that should apply to many serverless use cases, although your own specific security and compliance requirements should be well understood and might include more than we describe here.

- **One IAM Role per Function**

  Each and every Lambda function within your AWS account should have a 1:1 relationship with an IAM role. Even if multiple functions begin with exactly the same policy, always decouple your IAM roles so that you can ensure least privilege policies for the future of your function.

For example, if you shared the IAM role of a Lambda function that needed access to an AWS KMS key across multiple Lambda functions, then all of those functions would now have access to the same encryption key.

- **Temporary AWS Credentials**

  You should not have any long-lived AWS credentials included within your Lambda function code or configuration. (This is a great use for static code analysis tools to ensure it never occurs in your code base!) For most cases, the IAM execution role is all that's required to integrate with other AWS services. Simply create AWS service clients within your code through the AWS SDK without providing any credentials. The SDK automatically manages the retrieval and rotation of the temporary credentials generated for your role. The following is an example using Java.

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.defaultClient();
Table myTable = new Table(client, "MyTable");
```

This code snippet is all that's required for the AWS SDK for Java to create an object for interacting with a DynamoDB table that automatically sign its requests to the DynamoDB APIs, using the temporary IAM credentials assigned to your function.

However, there might be cases where the execution role is not sufficient for the type of access your function requires. This can be the case for some cross-account integrations your Lambda function might perform, or if you have user-specific access control policies through combining Amazon Cognito identity roles and DynamoDB fine-grained access control. For cross-account use cases, you should grant your execution role should be granted access to the AssumeRole API within the AWS Security Token Service and integrated to retrieve temporary access credentials.

For user-specific access control policies, your function should be provided with the user identity in question and then integrated with the Amazon Cognito API GetCredentialsForIdentity. In this case, it's imperative that you ensure your code appropriately manages these credentials so that you are leveraging the correct credentials for each user associated with that invocation of your Lambda function. It's common for an application to encrypt and store these per-user credentials in a place like DynamoDB or Amazon ElastiCache as part of user session data, so that they can be retrieved with reduced latency and more scalability than regenerating them for subsequent requests for a returning user.

- **Persisting Secrets**

  There are cases where you may have long-lived secrets (for example, database credentials, dependency service access keys, encryption keys, etc.) that your Lambda function needs to use. We recommend a few options for the lifecycle of secrets management in your application:

- Lambda Environment Variables with Encryption Helpers

  Advantages – Provided directly to your function runtime environment, minimizing the latency and code required to retrieve the secret.

  Disadvantages – Environment variables are coupled to a function version. Updating an environment variable requires a new function version (more rigid, but does provide stable version history as well).

- Amazon EC2 Systems Manager Parameter Store

  Advantages – Fully decoupled from your Lambda functions to provide maximum flexibility for how secrets and functions relate to each other.

  Disadvantages – A request to Parameter Store is required to retrieve a parameter/secret. While not substantial, this does add latency over environment variables as well as an additional service dependency, and requires writing slightly more code.

- **Using Secrets**

  Secrets should always only exist in memory and never be logged or written to disk. Write code that manages the rotation of secrets in the event a secret needs to be revoked while your application remains running.

- **API Authorization**

  Using API Gateway as the event source for your Lambda function is unique from the other AWS service event source options in that you have ownership of authentication and authorization of your API clients. API Gateway can perform much of the heavy lifting by providing things like native AWS SigV4 authentication, generated client SDKs, and custom authorizers. However, you're still responsible for ensuring that the security posture of your APIs meets the bar you've set. For more information about API security best practices, see this documentation.

- **VPC Security**

  If your Lambda function requires access to resources deployed inside a VPC, you should apply network security best practices through use of least privilege security groups, Lambda function-specific subnets, network ACLs, and route tables that allow traffic coming only from your Lambda functions to reach intended destinations.

  Keep in mind that these practices and policies impact the way that your Lambda functions connect to their dependencies. Invoking a Lambda function still occurs through event sources and the Invoke API (neither are affected by your VPC configuration).

- **Deployment Access Control**

  A call to the UpdateFunctionCode API is analogous to a code deployment. Moving an alias through the UpdateAlias API to that newly published version is analogous to a code release. Treat access to the Lambda APIs that enable function code/aliases with extreme sensitivity. As such, you should eliminate direct user access to these APIs for any functions (production functions at a minimum) to remove the possibility of human error. Making code changes to a Lambda function should be achieved through automation. With that in mind, the entry point for a deployment to Lambda becomes the place where your continuous integration/continuous delivery (CI/CD) pipeline is initiated. This may be a release branch in a repository, an S3 bucket where a new code package is uploaded that triggers an AWS CodePipeline pipeline, or somewhere else that's specific to your organization and processes. Wherever it is, it becomes a new place where you should enforce stringent access control mechanisms that fit your team structure and roles.

Reliability Best Practices

Serverless applications can be built to support mission-critical use cases. Just as with any mission-critical application, it's important that you architect with the mindset that Werner Vogels, CTO, Amazon.com, advocates for, "Everything fails all the time." For serverless applications, this could mean introducing logic bugs into your code, failing application dependencies, and other similar application-level issues that you should try and prevent and account for using existing best practices that will still apply to your serverless applications. For infrastructure-level service events, where you are abstracted away from the event for serverless applications, you should understand how you have architected your application to achieve high availability and fault tolerance.

High Availability

High-availability is important for production applications. The availability posture of your Lambda function depends on the number of Availability Zones it can be executed in. If your function uses the default network environment, it is automatically available to execute within all of the Availability Zones in that AWS Region. Nothing else is required to configure high availability for your function in the default network environment. If your function is deployed within your own VPC, the subnets (and their respective Availability Zones) define if your function remains available in the event of an Availability

Zone outage. Therefore, it's important that your VPC design includes subnets in multiple Availability Zones. In the event that an Availability Zone outage occurs, it's important that your remaining subnets continue to have adequate IP addresses to support the number of concurrent functions required. For information on how to calculate the number of IP addresses your functions require, see this documentation.

Fault Tolerance

If the application availability you need requires you to take advantage of multiple AWS Regions, you must take this into account up front in your design. It's not a complex exercise to replicate your Lambda function code packages to multiple AWS Regions. What can be complex, like most multi-region application designs, is coordinating a failover decision across all tiers of your application stack. This means you need to understand and orchestrate the shift to another AWS Region—not just for your Lambda functions but also for your event sources (and dependencies further upstream of your event sources) and persistence layers. In the end, a multi-region architecture is very application-specific. The most important thing to do to make a multi-region design feasible is to account for it in your design up front.

Recovery

Consider how your serverless application should behave in the event that your functions cannot be executed. For use cases where API Gateway is used as the event source, this can be as simple as gracefully handling error messages and providing a viable, if degraded, user experience until your functions can be successfully executed again.

For asynchronous use cases, it can be very important to still ensure that no function invocations are lost during the outage period. To ensure that all received events are processed after your function has recovered, you should take advantage of dead letter queues (p. 15) and implement how to process events placed on that queue after recovery occurs.

Performance Efficiency Best Practices

Before we dive into performance best practices, keep in mind that if your use case can be achieved asynchronously, you might not need to be concerned with the performance of your function (other than to optimize costs). You can leverage one of the event sources that will use the event **InvocationType** or use the pull-based invocation model. Those methods alone might allow your application logic to proceed while Lambda continues to process the event separately. If Lambda function execution time is something you want to optimize, the execution duration of your Lambda function will be primarily impacted by three things (in order of simplest to optimize): the resources you allocate in the function configuration, the language runtime you choose, and the code you write.

Choosing the Optimal Memory Size

Lambda provides a single dial to turn up and down the amount of compute resources available to your function—the amount of RAM allocated to your function. The amount of allocated RAM also impacts the amount of CPU time and network bandwidth your function receives. Simply choosing the smallest resource amount that runs your function adequately fast is an anti-pattern. Because Lambda is billed in 100-ms increments, this strategy might not only add latency to your application, it might even be more expensive overall if the added latency outweighs the resource cost savings.

We recommend that you test your Lambda function at each of the available resource levels to determine what the optimal level of price/performance is for your application. You'll discover that the performance of your function should improve logarithmically as resource levels are increased. The logic you're executing will define the lower bound for function execution time. There will also be a resource threshold where any additional RAM/CPU/bandwidth available to your function no longer provides any substantial performance gain. However, pricing increases linearly as the resource levels increase in Lambda. Your

tests should find where the logarithmic function bends to choose the optimal configuration for your function.

The following graph shows how the ideal memory allocation to an example function can allow for both better cost and lower latency. Here, the additional compute cost per 100 ms for using 512 MB over the lower memory options is outweighed by the amount of latency reduced in the function by allocating more resources. But after 512 MB, the performance gains are diminished for this particular function's logic, so the additional cost per 100 ms now drives the total cost higher. This leaves 512 MB as the optimal choice for minimizing total cost.

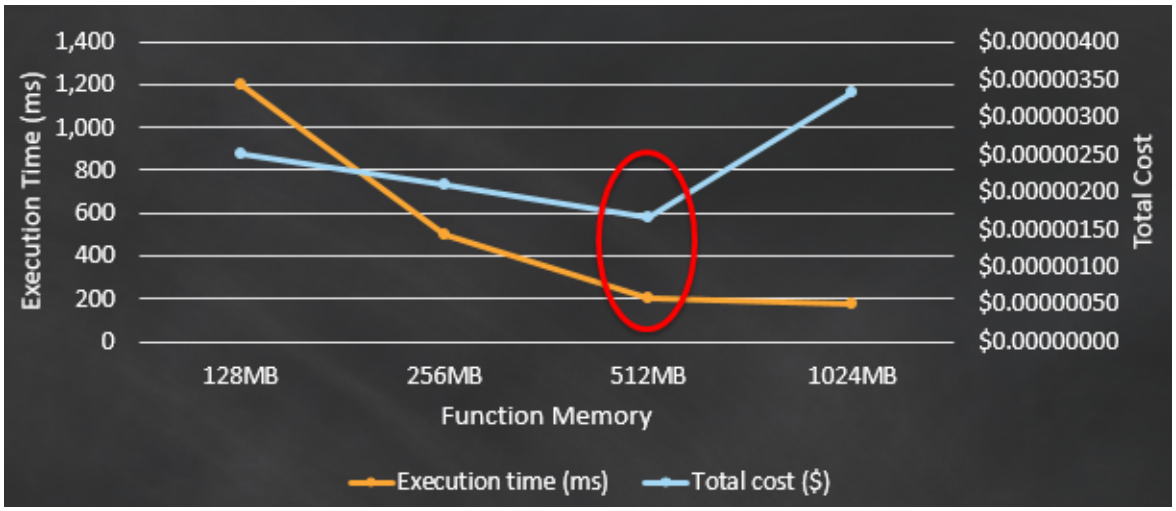

Figure 4: Choosing the optimal Lambda function memory size

The memory usage for your function is determined per invocation and can be viewed in CloudWatch Logs. On each invocation a REPORT: entry is made, as shown below.

```
REPORT RequestId: 3604209a-e9a3-11e6-939a-754dd98c7be3 Duration:
12.34 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory
Used: 18 MB
```

By analyzing the `Max Memory Used` field, you can determine if your function needs more memory or if you over-provisioned your function's memory size.

Language Runtime Performance

Choosing a language runtime performance is obviously dependent on your level of comfort and skills with each of the supported runtimes. But if performance is the driving consideration for your application, the performance characteristics of each language are what you might expect on Lambda as you would in another runtime environment: the compiled languages (Java and .NET) incur the largest initial startup cost for a container's first invocation, but show the best performance for subsequent invocations. The interpreted languages (Node.js and Python) have very fast initial invocation times compared to the compiled languages, but can't reach the same level of maximum performance as the compiled languages do.

If your application use case is both very latency-sensitive and susceptible to incurring the initial invocation cost frequently (very spiky traffic or very infrequent use), we recommend one of the interpreted languages.

If your application does not experience large peaks or valleys within its traffic patterns, or does not have user experiences blocked on Lambda function response times, we recommend you choose the language you're already most comfortable with.

Optimizing Your Code

Much of the performance of your Lambda function is dictated by what logic you need your Lambda function to execute and what its dependencies are. We won't cover what all those optimizations could be, because they vary from application to application. But there are some general best practices to optimize your code for Lambda. These are related to taking advantage of container reuse (as describes in the previous overview) and minimizing the initial cost of a cold start.

Here are a few examples of how you can improve the performance of your function code when a warm container is invoked:

- After initial execution, store and reference any externalized configuration or dependencies that your code retrieves locally.
- Limit the reinitialization of variables/objects on every invocation (use global/static variables, singletons, etc.).
- Keep alive and reuse connections (HTTP, database, etc.) that were established during a previous invocation.

Finally, you should do the following to limit the amount of time that a cold start takes for your Lambda function:

1. Always use the default network environment unless connectivity to a resource within a VPC via private IP is required. This is because there are additional cold start scenarios related to the VPC configuration of a Lambda function (related to creating ENIs within your VPC).
2. Choose an interpreted language over a compiled language.
3. Trim your function code package to only its runtime necessities. This reduces the amount of time that it takes for your code package to be downloaded from Amazon S3 ahead of invocation.

Understanding Your Application Performance

To get visibility into the various components of your application architecture, which could include one or more Lambda functions, we recommend that you use AWS X-Ray. X-Ray lets you trace the full lifecycle of an application request through each of its component parts, showing the latency and other metrics of each component separately, as shown in the following figure.

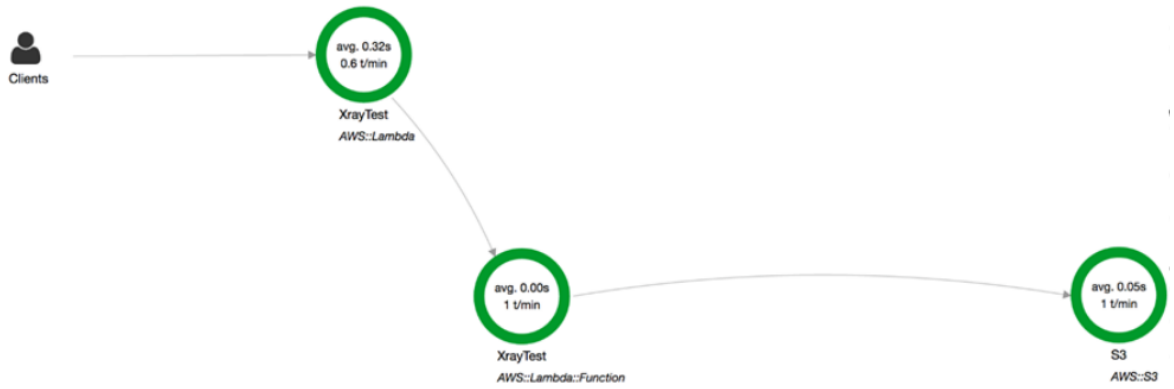

Figure 5: A service map visualized by AWS X-Ray

To learn more about X-Ray, see this documentation.

Operational Excellence Best Practices

Creating a serverless application removes many operational burdens that a traditional application brings with it. This doesn't mean you should reduce your focus on operational excellence. It means that you can narrow your operational focus to a smaller number of responsibilities and hopefully achieve a higher level of operational excellence.

Logging

Each language runtime for Lambda provides a mechanism for your function to deliver logged statements to CloudWatch Logs. Making adequate use of logs goes without saying and isn't new to Lambda and serverless architectures. Even though it's not considered best practice today, many operational teams depend on viewing logs as they are generated on top of the server an application is deployed on. That simply isn't possible with Lambda because there is no server. You also don't have the ability to "step through" the code of a live running Lambda function today (although you can do this with AWS SAM Local prior to deployment). For deployed functions, you depend heavily on the logs you create to inform an investigation of function behavior. Therefore, it's especially important that the logs you do create find the right balance of verbosity to help track/triage issues as they occur without demanding too much additional compute time to create them.

We recommend that you make use of Lambda environment variables to create a LogLevel variable that your function can refer to so that it can determine which log statements to create during runtime. Appropriate use of log levels can ensure that you have the ability to selectively incur the additional compute cost and storage cost only during an operational triage.

Metrics and Monitoring

Lambda, just like other AWS services, provides a number of CloudWatch metrics out of the box. These include metrics related to the number of invocations a function has received, the execution duration of a function, and others. It's best practice to create alarm thresholds (high and low) for each of your Lambda functions on *all* of the provided metrics through CloudWatch. A major change in how your function is invoked or how long it takes to execute could be your first indication of a problem in your architecture.

For any additional metrics that your application needs to gather (for example, application error codes, dependency-specific latency, etc.) you have two options to get those custom metrics stored in CloudWatch or your monitoring solution of choice:

- Create a custom metric and integrate directly with the API required from your Lambda function as it's executing. This has the fewest dependencies and will record the metric as fast as possible. However, it does require you to spend Lambda execution time and resources integrating with another service dependency. If you follow this path, ensure that your code for capturing metrics is modularized and reusable across your Lambda functions instead of tightly coupled to a specific Lambda function.
- Capture the metric within your Lambda function code and log it using the provided logging mechanisms in Lambda. Then, create a CloudWatch Logs metric filter on the function streams to extract the metric and make it available in CloudWatch. Alternatively, create another Lambda function as a subscription filter on the CloudWatch Logs stream to push filtered log statements to another metrics solution. This path introduces more complexity and is not as near real-time as the previous solution for capturing metrics. However, it allows your function to more quickly create metrics through logging rather than making an external service request.

Deployment

Performing a deployment in Lambda is as simple as uploading a new function code package, publishing a new version, and updating your aliases. However, these steps should only be pieces of your deployment process with Lambda. Each deployment process is application-specific. To design a deployment process that avoids negatively disrupting your users or application behavior, you need to understand the

relationship between each Lambda function and its event sources and dependencies. Things to consider are:

- **Parallel version invocations** – Updating an alias to point to a new version of a Lambda function happens asynchronously on the service side. There will be a short period of time that existing function containers containing the previous source code package will continue to be invoked alongside the new function version the alias has been updated to. It's important that your application continues to operate as expected during this process. An artifact of this might be that any stack dependencies being decommissioned after a deployment (for example, database tables, a message queue, etc.) not be decommissioned until after you've observed all invocations targeting the new function version.
- **Deployment schedule** – Performing a Lambda function deployment during a peak traffic time could result in more cold start times than desired. You should always perform your function deployments during a low traffic period to minimize the immediate impact of the new/cold function containers being provisioned in the Lambda environment.
- **Rollback** – Lambda provides details about Lambda function versions (for example, created time, incrementing numbers, etc.). However, it doesn't logically track how your application lifecycle has been using those versions. If you need to roll back your Lambda function code, it's important for your processes to roll back to the function version that was previously deployed.

Load Testing

Load test your Lambda function to determine an optimum timeout value. It's important to analyze how long your function runs so that you can better determine any problems with a dependency service that might increase the concurrency of the function beyond what you expect. This is especially important when your Lambda function makes network calls to resources that may not handle Lambda's scaling.

Triage and Debugging

Both logging to enable investigations and using X-Ray to profile applications are useful to operational triages. Additionally, consider creating Lambda function aliases that represent operational activities such as integration testing, performance testing, debugging, etc. It's common for teams to build out test suites or segmented application stacks that serve an operational purpose. You should build these operational artifacts to also integrate with Lambda functions via aliases. However, keep in mind that aliases don't enforce a wholly separate Lambda function container. So an alias like PerfTest that points at function version number N, will use the same function containers as all other aliases pointing at version N. You should define appropriate versioning and alias updating processes to ensure separate containers are invoked where required.

Cost Optimization Best Practices

Because Lambda charges are based on function execution time and the resources allocated, optimizing your costs is focused on optimizing those two dimensions.

Right-Sizing

As covered in Performance Efficiency (p. 19), it's an anti-pattern to assume that the smallest resource size available to your function will provide the lowest total cost. If your function's resource size is too small, you could pay more due to a longer execution time than if more resources were available that allowed your function to complete more quickly.

See the section Choosing the Optimal Memory Size (p. 19) for more details.

Distributed and Asynchronous Architectures

You don't need to implement all use cases through a series of blocking/synchronous API requests and responses. If you are able to design your application to be asynchronous, you might find that each

decoupled component of your architecture takes less compute time to conduct its work than tightly coupled components that spend CPU cycles awaiting responses to synchronous requests. Many of the Lambda event sources fit well with distributed systems and can be used to integrate your modular and decoupled functions in a more cost-effective manner.

Batch Size

Some Lambda event sources allow you to define the batch size for the number of records that are delivered on each function invocation (for example, Kinesis and DynamoDB). You should test to find the optimal number of records for each batch size so that the polling frequency of each event source is tuned to how quickly your function can complete its task.

Event Source Selection

The variety of event sources available to integrate with Lambda means that you often have a variety of solution options available to meet your requirements. Depending on your use case and requirements (request scale, volume of data, latency required, etc.), there might be a non-trivial difference in the total cost of your architecture based on which AWS services you choose as the components that surround your Lambda function.

Serverless Development Best Practices

Creating applications with Lambda can enable a development pace that you haven't experienced before. The amount of code you need to write for a working and robust serverless application will likely be a small percentage of the code you would need to write for a server-based model. But with a new application delivery model that serverless architectures enable, there are new dimensions and constructs that your development processes must make decisions about. Things like organizing your code base with Lambda functions in mind, moving code changes from a developer laptop into a production serverless environment, and ensuring code quality through testing even though you can't simulate the Lambda runtime environment or your event sources outside of AWS. The following are some development-centric best practices to help you work through these aspects of owning a serverless application.

Infrastructure as Code – the AWS Serverless Application Model (AWS SAM)

Representing your infrastructure as code brings many benefits in terms of the auditability, automatability, and repeatability of managing the creation and modification of infrastructure. Even though you don't need to manage any infrastructure when building a serverless application, many components play a role in the architecture: IAM roles, Lambda functions and their configurations, their event sources, and other dependencies. Representing all of these things in AWS CloudFormation natively would require a large amount of JSON or YAML. Much of it would be almost identical from one serverless application to the next.

The AWS Serverless Application Model (AWS SAM) enables you to have a simpler experience when building serverless applications and get the benefits of infrastructure as code. AWS SAM is an open specification abstraction layer on top of AWS CloudFormation. It provides a set of command line utilities that enable you to define a full serverless application stack with only a handful of lines of JSON or YAML, package your Lambda function code together with that infrastructure definition, and then deploy them together to AWS. We recommend using AWS SAM combined with AWS CloudFormation to define and make changes to your serverless application environment.

There is a distinction, however, between changes that occur at the infrastructure/environment level and application code changes occurring within existing Lambda functions. AWS CloudFormation and

AWS SAM aren't the only tools required to build a deployment pipeline for your Lambda function code changes. See the CI/CD section (p. 28) of this whitepaper for more recommendations about managing code changes for your Lambda functions.

Local Testing – AWS SAM Local

Along with AWS SAM, AWS SAM Local offers additional command line tools that you can add to AWS SAM to test your serverless functions and applications locally before deploying them to AWS. AWS SAM Local uses Docker to enable you to quickly test your developed Lambda functions using popular event sources (for example, Amazon S3, DynamoDB, etc.). You can locally test an API you define in your SAM template before it is created in API Gateway. You can also validate the AWS SAM template that you created. By enabling these capabilities to run against Lambda functions still residing within your developer workstation, you can do things like view logs locally, step through your code in a debugger, and quickly iterate changes without having to deploy a new code package to AWS.

Coding and Code Management Best Practices

When developing code for Lambda functions, there are some specific recommendations around how you should both write and organize code so that managing many Lambda functions doesn't become a complex task.

Coding Best Practices

Depending on the Lambda runtime language you build with, continue to follow the best practices already established for that language. While the environment that surrounds how your code is invoked has changed with Lambda, the language runtime environment is the same as anywhere else. Coding standards and best practices still apply. The following recommendations are specific to writing code for Lambda, outside of those general best practices for your language of choice.

Business Logic outside the Handler

Your Lambda function starts execution at the handler function (p. 5) you define within your code package. Within your handler function you should receive the parameters provided by Lambda, pass those parameters to another function to parse into new variables/objects that are contextualized to your application, and then reach out to your business logic that sits outside the handler function and file. This enables you to create a code package that is as decoupled from the Lambda runtime environment as possible. This will greatly benefit your ability to test your code within the context of objects and functions you've created and reuse the business logic you've written in other environments outside of Lambda.

The following example (written in Java) shows poor practices where the core business logic of an application is tightly coupled to Lambda. In this example, the business logic is created within the handler method and depends directly on Lambda event source objects.

```
//Poor example of best practices, business logic is tightly coupled to Lambda environment
public class LambdaFunctionHandler {

    public String handleRequest(S3Event event, Context context) {
        context.getLogger().log("Starting execution.");

        AmazonDynamoDB client = AmazonDynamoDBClientBuilder.defaultClient();
        Table myTable = new Table(client, "MyTable");

        /**
         * The simple business logic below (marking items processed in DDB)
         * is directly dependent on the structure of the Lambda events
         * that S3 sends.  Performing unit tests of core business logic
         * will require me to mock an S3Event object and the Lambda service
         * created Context object in order to invoke this method.
         */
        for (S3EventNotificationRecord e : event.getRecords()) {
            context.getLogger().log("Processing records.");
            PrimaryKey key = new PrimaryKey("S3ObjectKey", e.getS3().getObject().getKey());
            Item item = new Item().withPrimaryKey(key).withBoolean("Processed", true);
            myTable.putItem(item);
        }

        context.getLogger().log("Completing execution");
        return "{\"status\": \"succes\"}";

    }

}
```

Warm Containers—Caching/Keep-Alive/Reuse

As mentioned earlier (p. 7), you should write code that takes advantage of a warm function container. This means scoping your variables in a way that they and their contents can be reused on subsequent invocations where possible. This is especially impactful for things like bootstrapping configuration, keeping external dependency connections open, or one-time initialization of large objects that can persist from one invocation to the next.

Control Dependencies

The Lambda execution environment contains many libraries such as the AWS SDK for the Node.js and Python runtimes. (For a full list, see the Lambda Execution Environment and Available Libraries.) To enable the latest set of features and security updates, Lambda periodically updates these libraries. These updates can introduce subtle changes to the behavior of your Lambda function. To have full control of the dependencies your function uses, we recommend packaging all your dependencies with your deployment package.

Trim Dependencies

Lambda function code packages are permitted to be at most 50 MB when compressed and 250 MB when extracted in the runtime environment. If you are including large dependency artifacts with your function code, you may need to trim the dependencies included to just the runtime essentials. This also allows your Lambda function code to be downloaded and installed in the runtime environment more quickly for cold starts.

Fail Fast

Configure reasonably short timeouts for any external dependencies, as well as a reasonably short overall Lambda function timeout. Don't allow your function to spin helplessly while waiting for a dependency to

respond. Because Lambda is billed based on the duration of your function execution, you don't want to incur higher charges than necessary when your function dependencies are unresponsive.

Handling Exceptions

You might decide to throw and handle exceptions differently depending on your use case for Lambda. If you're placing an API Gateway API in front of a Lambda function, you may decide to throw an exception back to API Gateway where it might be transformed, based on its contents, into the appropriate HTTP status code and message for the error that occurred. If you're building an asynchronous data processing system, you might decide that some exceptions within your code base should equate to the invocation moving to the dead letter queue for reprocessing, while other errors can just be logged and not placed on the dead letter queue. You should evaluate what your decide failure behaviors are and ensure that you are creating and throwing the correct types of exceptions within your code to achieve that behavior. To learn more about handling exceptions, see the following for details about how exceptions are defined for each language runtime environment:

- Java
- Node.js
- Python
- C#

Code Management Best Practices

Now that the code you've written for your Lambda functions follows best practices, how should you manage that code? With the development speed that Lambda enables, you might be able to complete code changes at a pace that is unfamiliar for your typical processes. And the reduced amount of code that serverless architectures require means that your Lambda function code represents a large portion of what makes your entire application stack function. So having good source code management of your Lambda function code will help ensure secure, efficient, and smooth change management processes.

Code Repository Organization

We recommend that you organize your Lambda function source code to be very fine-grained within your source code management solution of choice. This usually means having a 1:1 relationship between Lambda functions and code repositories or repository projects. (The lexicon differs from one source code management tool to another.) However, if you are following a strategy of creating separate Lambda functions for different lifecycle stages of the same logical function (that is, you have two Lambda functions, one called MyLambdaFunction-DEV and another called MyLambdaFunction-PROD), it makes sense to have those separate Lambda functions share a code base (perhaps deploying from separate release branches).

The main purpose of organizing your code this way is to help ensure that all of the code that contributes to the code package of a particular Lambda function is independently versioned and committed to, and defines its own dependencies and those dependencies' versions. Each Lambda function should be fully decoupled from a source code perspective from other Lambda functions, just as it will be when it's deployed. You don't want to go through the process of modernizing an application architecture to be modular and decoupled with Lambda only to be left with a monolithic and tightly coupled code base.

Release Branches

We recommend that you create a repository or project branching strategy that enables you to correlate Lambda function deployments with incremental commits on a release branch. If you don't have a way to confidently correlate source code changes within your repository and the changes that have been deployed to a live Lambda function, an operational investigation will always begin with trying to identify which version of your code base is the one currently deployed. You should build a CI/CD pipeline

(more recommendations for this later) that allows you to correlate Lambda code package creation and deployment times with the code changes that have occurred with your release branch for that Lambda function.

Testing

Time spent developing thorough testing of your code is the best way to ensure quality within a serverless architecture. However, serverless architectures will enforce proper unit testing practices perhaps more than you're used to. Many developers use unit test tools and frameworks to write tests that cause their code to also test its dependencies. This is a single test that combines a unit test and an integration test but that doesn't perform either very well.

It's important to scope all of your unit test cases down to a single code path within a single logical function, mocking all inputs from upstream and outputs from downstream. This allows you to isolate your test cases to only the code that you own. When writing unit tests, you can and should assume that your dependencies behave properly based on the contracts your code has with them as APIs, libraries, etc.

It's similarly important for your integration tests to test the integration of your code to its dependencies in an environment that mimics the live environment. Testing whether a developer laptop or build server can integrate with a downstream dependency isn't fully testing if your code will integrate successfully once in the live environment. This is especially true of the Lambda environment, where you code doesn't have ownership of the events that are going to be delivered by event sources and you don't have the ability to create the Lambda runtime environment outside of Lambda.

Unit Tests

With what we've said earlier in mind, we recommend that you unit test your Lambda function code thoroughly, focusing mostly on the business logic outside your handler function. You should also unit test your ability to parse sample/mock objects for the event sources of your function. However, the bulk of your logic and tests should occur with mocked objects and functions that you have full control over within your code base. If you feel that there are important things inside your handler function that need to be unit tested, it can be a sign you should encapsulate and externalize the logic in your handler function further. Also, to supplement the unit tests you've written, you should create local test automation using AWS SAM Local that can serve as local end-to-end testing of your function code (note that this isn't a replacement for unit testing).

Integration Testing

For integration tests, we recommend that you create lower lifecycle versions of your Lambda functions where your code packages are deployed and invoked through sample events that your CI/CD pipeline can trigger and inspect the results of. (Implementation depends on your application and architecture.)

Continuous Delivery

We recommend that you programmatically manage all of your serverless deployments through CI/CD pipelines. This is because the speed with which you will be able to develop new features and push code changes with Lambda will allow you to deploy much more frequently. Manual deployments, combined with a need to deploy more frequently, often result in both the manual process becoming a bottleneck and prone to error.

The capabilities provided by AWS CodeCommit, AWS CodePipeline, AWS CodeBuild, AWS SAM, and AWS CodeStar provide a set of capabilities that you can natively combine into a holistic and automated serverless CI/CD pipeline (where the pipeline itself also has no infrastructure that you need to manage).

Here is how each of these services plays a role in a well-defined continuous delivery strategy.

AWS CodeCommit– Provides hosted private Git repositories that will enable you to host your serverless source code, create a branching strategy that meets our recommendations (including fine-grained access control), and integrate with AWS CodePipeline to trigger a new pipeline execution when a new commit occurs in your release branch.

AWS CodePipeline – Defines the steps in your pipeline. Typically an AWS CodePipeline pipeline begins where your source code changes arrive. Then you execute a build phase, execute tests against your new build, and perform a deployment and release of your build into the live environment. AWS CodePipeline provides native integration options for each of these phases with other AWS services.

AWS CodeBuild – Can be used for the build state of your pipeline. Use it to build your code, execute unit tests, and create a new Lambda code package. Then, integrate with AWS SAM to push your code package to Amazon S3 and push the new package to Lambda via AWS CloudFormation.

After your new version is published to your Lambda function through AWS CodeBuild, you can automate your subsequent steps in your AWS CodePipeline pipeline by creating deployment-centric Lambda functions. They will own the logic for performing integration tests, updating function aliases, determining if immediate rollbacks are necessary, and any other application-centric steps needed to occur during a deployment for your application (like cache flushes, notification messages, etc.). Each one of these deployment-centric Lambda functions can be invoked in sequence as a step within your AWS CodePipeline pipeline using the Invoke action. For details on using Lambda within AWS CodePipeline, see this documentation.

In the end, each application and organization has its own requirements for moving source code from repository to production. The more automation you can introduce into this process, the more agility you can achieve using Lambda.

AWS CodeStar – A unified user interface for creating a serverless application (and other types of applications) that helps you follow these best practices from the beginning. When you create a new project in AWS CodeStar, you automatically begin with a fully implemented and integrated continuous delivery toolchain (using AWS CodeCommit, AWS CodePipeline, and AWS CodeBuild services mentioned earlier). You will also have a place where you can manage all aspects of the SDLC for your project, including team member management, issue tracking, development, deployment, and operations. For more information about AWS CodeStar, go here.

Sample Serverless Architectures

There are a number of sample serverless architectures and instructions for recreating them in your own AWS account. You can find them on GitHub.

Conclusion

Building serverless applications on AWS relieves you of the responsibilities and constraints that servers introduce. Using AWS Lambda as your serverless logic layer enables you to build faster and focus your development efforts on what differentiates your application. Alongside Lambda, AWS provides additional serverless capabilities so that you can build robust, performant, event-driven, reliable, secure, and cost-effective applications. Understanding the capabilities and recommendations described in this whitepaper can help ensure your success when building serverless applications of your own. To learn more on related topics, see Serverless Computing and Applications.

Contributors

The following individuals and organizations contributed to this document:

- Andrew Baird, Sr. Solutions Architect, AWS
- George Huang, Sr. Product Marketing Manager, AWS
- Chris Munns, Sr. Developer Advocate, AWS
- Orr Weinstein, Sr. Product Manager, AWS

Document Revisions

To be notified about updates to this whitepaper, subscribe to the RSS feed.

update-history-change	update-history-description	update-history-date
Initial publication (p. 33)	Whitepaper first published	November 1, 2017

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.