
AWS Lambda

Developer Guide



AWS Lambda: Developer Guide

Copyright © 2019 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What Is AWS Lambda?	1
When Should I Use AWS Lambda?	1
Are You a First-time User of AWS Lambda?	2
Getting Started	3
Create a Function	3
Use the Designer	3
Invoke the Lambda Function	4
Concepts	5
Command Line Tools	6
Set Up the AWS CLI	6
AWS Serverless Application Model CLI	6
Limits	7
Permissions	9
Execution Role	9
Resource-based Policies	10
Granting Function Access to AWS Services	11
Granting Function Access to Other Accounts	12
Granting Layer Access to Other Accounts	12
Cleaning up Resource-based Policies	13
User Policies	13
Function Development	14
Layer Development and Use	17
Cross-Account Roles	18
Resources and Conditions	18
Functions	19
Event Source Mappings	21
Layers	21
Lambda Functions	22
Building Lambda Functions	22
Authoring Code for Your Lambda Function	22
Deploying Code and Creating a Lambda Function	23
Monitoring and Troubleshooting	24
Code Editor	24
Working with Files and Folders	25
Working with Code	27
Using the Menu Bar	29
Working in Fullscreen Mode	30
Working with Preferences	30
Working with Commands	30
Programming Model	31
Deployment Package	32
Permissions Policies on Lambda Deployment Packages	32
Accessing AWS Resources	33
Accessing AWS Services	33
Accessing non AWS Services	33
Accessing Private Services or Resources	34
Configuring Functions	35
Function Configuration	35
Managing Concurrency	37
Account Level Concurrent Execution Limit	37
Function Level Concurrent Execution Limit	37
Throttling Behavior	39
Monitoring Your Concurrency Usage	39
Environment Variables	40

Setting Up	40
Rules for Naming Environment Variables	42
Environment Variables and Function Versioning	43
Environment Variable Encryption	43
Create a Lambda Function Using Environment Variables	44
Create a Lambda Function Using Environment Variables To Store Sensitive Information	45
Versioning	47
Versioning	48
Aliases	51
Resource Policies	58
Managing Versioning	59
Traffic Shifting Using Aliases	62
Layers	64
Configuring a Function to Use Layers	64
Managing Layers	65
Including Library Dependencies in a Layer	67
Layer Permissions	68
VPC Settings	68
Configuring a Lambda Function for Amazon VPC Access	68
Internet Access for Lambda Functions	69
Guidelines for Setting Up VPC-Enabled Lambda Functions	70
Tutorial: Amazon ElastiCache	70
Tutorial: Amazon RDS	73
Tagging	76
Tagging Lambda Functions for Billing	76
Applying Tags to Lambda Functions Using the Console	77
Applying Tags to Lambda Functions Using the CLI	77
Filtering on Tagged Lambda Functions	78
Tag Restrictions	79
Invoking Functions	80
Example 1: Amazon S3 Pushes Events and Invokes a Lambda Function	80
Example 2: AWS Lambda Pulls Events from a Kinesis Stream and Invokes a Lambda Function	81
Invocation Types	82
Event Source Mapping	82
Event Source Mapping for AWS Services	83
Event Source Mapping for AWS Poll-Based Services	84
Event Source Mapping for Custom Applications	84
Retry Behavior	85
Scaling	86
Request Rate	87
Automatic Scaling	87
Dead Letter Queues	88
AWS CLI	89
Prerequisites	90
Create the Execution Role	90
Create the Function	90
List the Lambda Functions in Your Account	91
Clean Up	93
Mobile SDK for Android	93
Tutorial	93
Sample Code	99
Lambda Runtimes	102
Environment Variables	103
Execution Context	104
Runtime Support Policy	105
Custom Runtimes	106
Using a Custom Runtime	106

Building a Custom Runtime	107
Runtime Interface	108
Next Invocation	109
Invocation Response	109
Invocation Error	110
Initialization Error	110
Tutorial – Custom Runtime	110
Prerequisites	111
Create a Function	111
Create a Layer	113
Update the Function	113
Update the Runtime	114
Share the Layer	115
Clean Up	115
Lambda Applications	116
Manage Applications	116
Monitoring Applications	117
Custom Monitoring Dashboards	117
AWS SAM	119
Sample – Error Processor	119
Architecture and Event Structure	120
Instrumentation with AWS X-Ray	121
AWS CloudFormation Template and Additional Resources	122
Continuous Delivery	123
Prerequisites	123
Create an AWS CloudFormation Role	124
Set Up a Repository	124
Create a Pipeline	126
Update the Build Stage Role	127
Complete the Deployment Stage	127
Best Practices	127
Function Code	128
Function Configuration	129
Alarming and Metrics	129
Stream Event Invokes	129
Async Invokes	130
Lambda VPC	130
Working with Other Services	132
Application Load Balancer	133
Alexa	135
Amazon API Gateway	135
Tutorial	137
Sample Code	146
Microservice Blueprint	148
Sample Template	149
AWS CloudTrail	150
Tutorial	151
Sample Code	157
CloudWatch Events	158
Tutorial	159
Sample Template	162
Schedule Expressions	163
Amazon CloudWatch Logs	164
AWS CloudFormation	164
CloudFront	166
AWS CodeCommit	168
Amazon Cognito	168

AWS Config	169
Amazon DynamoDB	170
Creating an Event Source Mapping	171
Execution Role Permissions	172
Tutorial	172
Sample Code	176
Sample Template	179
Kinesis	180
Configuring Your Data Stream and Function	181
Creating an Event Source Mapping	182
Event Source Mapping API	183
Execution Role Permissions	184
Amazon CloudWatch Metrics	184
Tutorial	184
Sample Code	188
Sample Template	191
Kinesis Data Firehose	192
Amazon Lex	193
Amazon S3	194
Tutorial	196
Sample Code	202
Sample Template	208
Amazon SES	209
Amazon SNS	211
Tutorial	212
Sample Code	214
Amazon SQS	216
Configuring a Queue for Use With Lambda	218
Configuring a Queue as an Event Source	218
Execution Role Permissions	218
Tutorial	219
Sample Code	222
Sample Template	224
Monitoring	226
Using Amazon CloudWatch	226
AWS Lambda Troubleshooting Scenarios	226
Monitoring Graphs	228
Accessing CloudWatch Logs	229
Lambda Metrics	229
Using AWS X-Ray	232
Tracing Lambda-Based Applications with AWS X-Ray	232
Setting Up AWS X-Ray with Lambda	234
Emitting Trace Segments from a Lambda Function	235
The AWS X-Ray Daemon in the Lambda Environment	236
Using Environment Variables to Communicate with AWS X-Ray	236
Lambda Traces in the AWS X-Ray Console: Examples	236
Using CloudTrail	238
AWS Lambda Information in CloudTrail	238
Understanding AWS Lambda Log File Entries	239
Using CloudTrail to Track Function Invocations	240
Working with Node.js	241
Deployment Package	241
Handler	242
Using the Callback Parameter	243
Example	244
Context	244
Logging	245

Errors	246
Function Error Handling	248
Tracing	249
Working with Python	251
Deployment Package	251
Without Additional Dependencies	252
With Additional Dependencies	252
With a Virtual Environment	253
Handler	255
Context	256
Logging	257
Errors	258
Function Error Handling	260
Tracing	261
Working with Java	263
Deployment Package	264
Maven	264
Maven and Eclipse	266
Gradle and ZIP	268
Eclipse IDE and AWS SDK Plugin	271
Handler	271
Handler Overload Resolution	272
Additional Information	273
Handler Input/Output Types (Java)	273
Leveraging Predefined Interfaces for Creating Handler (Java)	277
Context	281
Logging	281
LambdaLogger	282
Custom Appender for Log4j 2	283
Errors	284
Function Error Handling	285
Tracing	286
Sample Function	288
Working with Go	290
Deployment Package	290
Creating a Deployment Package on Windows	291
Handler	291
Handler	292
Using Global State	294
Context	295
Accessing Invoke Context Information	295
Logging	296
Errors	298
Function Error Handling	260
Handling Unexpected Errors	299
Tracing	300
Installing the X-Ray SDK for Go	300
Configuring the X-Ray SDK for Go	300
Create a subsegment	300
Capture	301
Tracing HTTP Requests	301
Environment Variables	301
Working with C#	302
Deployment Package	302
.NET Core CLI	303
AWS Toolkit for Visual Studio	310
Handler	311

Handling Streams	311
Handling Standard Data Types	312
Handler Signatures	313
Lambda Function Handler Restrictions	313
Using Async in C# Functions with AWS Lambda	314
Context	315
Logging	315
Errors	317
Function Error Handling	319
Working with PowerShell	321
Deployment Package	321
Setting Up a PowerShell Development Environment	322
Using the AWSLambdaPSCore Module	322
Handler	324
Returning Data	325
Context	325
Logging	326
Errors	327
Function Error Handling	328
Working with Ruby	329
Handler	330
Deployment Package	331
Updating a Function with No Dependencies	331
Updating a Function with Additional Dependencies	332
Context	333
Logging	334
Errors	335
API Reference	337
Actions	337
AddLayerVersionPermission	339
AddPermission	343
CreateAlias	347
CreateEventSourceMapping	351
CreateFunction	355
DeleteAlias	363
DeleteEventSourceMapping	365
DeleteFunction	368
DeleteFunctionConcurrency	370
DeleteLayerVersion	372
GetAccountSettings	374
GetAlias	376
GetEventSourceMapping	379
GetFunction	382
GetFunctionConfiguration	385
GetLayerVersion	390
GetLayerVersionByArn	393
GetLayerVersionPolicy	396
GetPolicy	398
Invoke	400
InvokeAsync	405
ListAliases	407
ListEventSourceMappings	410
ListFunctions	413
ListLayers	416
ListLayerVersions	418
ListTags	421
ListVersionsByFunction	423

PublishLayerVersion	426
PublishVersion	430
PutFunctionConcurrency	436
RemoveLayerVersionPermission	439
RemovePermission	441
TagResource	444
UntagResource	446
UpdateAlias	448
UpdateEventSourceMapping	452
UpdateFunctionCode	456
UpdateFunctionConfiguration	463
Data Types	470
AccountLimit	471
AccountUsage	472
AliasConfiguration	473
AliasRoutingConfiguration	475
Concurrency	476
DeadLetterConfig	477
Environment	478
EnvironmentError	479
EnvironmentResponse	480
EventSourceMappingConfiguration	481
FunctionCode	483
FunctionCodeLocation	484
FunctionConfiguration	485
Layer	489
LayersListItem	490
LayerVersionContentInput	491
LayerVersionContentOutput	492
LayerVersionsListItem	493
TracingConfig	495
TracingConfigResponse	496
VpcConfig	497
VpcConfigResponse	498
Certificate Errors When Using an SDK	498
Releases	500
Earlier Updates	502
AWS Glossary	507

What Is AWS Lambda?

AWS Lambda is a compute service that lets you run code without provisioning or managing servers. AWS Lambda executes your code only when needed and scales automatically, from a few requests per day to thousands per second. You pay only for the compute time you consume - there is no charge when your code is not running. With AWS Lambda, you can run code for virtually any type of application or backend service - all with zero administration. AWS Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, code monitoring and logging. All you need to do is supply your code in one of the [languages that AWS Lambda supports \(p. 102\)](#).

You can use AWS Lambda to run your code in response to events, such as changes to data in an Amazon S3 bucket or an Amazon DynamoDB table; to run your code in response to HTTP requests using Amazon API Gateway; or invoke your code using API calls made using AWS SDKs. With these capabilities, you can use Lambda to easily build data processing triggers for AWS services like Amazon S3 and Amazon DynamoDB, process streaming data stored in Kinesis, or create your own back end that operates at AWS scale, performance, and security.

You can also build [serverless](#) applications composed of functions that are triggered by events and automatically deploy them using CodePipeline and AWS CodeBuild. For more information, see [AWS Lambda Applications \(p. 116\)](#).

For more information about the AWS Lambda execution environment, see [AWS Lambda Runtimes \(p. 102\)](#). For information about how AWS Lambda determines compute resources required to execute your code, see [AWS Lambda Function Configuration \(p. 35\)](#).

When Should I Use AWS Lambda?

AWS Lambda is an ideal compute platform for many application scenarios, provided that you can write your application code in languages supported by AWS Lambda, and run within the AWS Lambda standard runtime environment and resources provided by Lambda.

When using AWS Lambda, you are responsible only for your code. AWS Lambda manages the compute fleet that offers a balance of memory, CPU, network, and other resources. This is in exchange for flexibility, which means you cannot log in to compute instances, or customize the operating system or language runtime. These constraints enable AWS Lambda to perform operational and administrative activities on your behalf, including provisioning capacity, monitoring fleet health, applying security patches, deploying your code, and monitoring and logging your Lambda functions.

If you need to manage your own compute resources, Amazon Web Services also offers other compute services to meet your needs.

- Amazon Elastic Compute Cloud (Amazon EC2) service offers flexibility and a wide range of EC2 instance types to choose from. It gives you the option to customize operating systems, network and security settings, and the entire software stack, but you are responsible for provisioning capacity, monitoring fleet health and performance, and using Availability Zones for fault tolerance.
- Elastic Beanstalk offers an easy-to-use service for deploying and scaling applications onto Amazon EC2 in which you retain ownership and full control over the underlying EC2 instances.

Lambda is a highly available service. For more information, see the [AWS Lambda Service Level Agreement](#).

Are You a First-time User of AWS Lambda?

If you are a first-time user of AWS Lambda, we recommend that you read the following sections in order:

1. **Read the product overview and watch the introductory video to understand sample use cases.** These resources are available on the [AWS Lambda webpage](#).
2. **Review the Lambda Functions (p. 22) section of this guide.** To understand the programming model and deployment options for a Lambda function there are core concepts you should be familiar with. This section explains these concepts and provides details of how they work in different languages that you can use to author your Lambda function code.
3. **Try the console-based Getting Started exercise.** The exercise provides instructions for you to create and test your first Lambda function using the console. You also learn about the console provided blueprints to quickly create your Lambda functions. For more information, see [Getting Started with AWS Lambda \(p. 3\)](#).
4. **Read the Deploying Applications with AWS Lambda (p. 116) section of this guide.** This section introduces various AWS Lambda components you work with to create an end-to-end experience.

Beyond the Getting Started exercise, you can explore the various use cases, each of which is provided with a tutorial that walks you through an example scenario. Depending on your application needs (for example, whether you want event driven Lambda function invocation or on-demand invocation), you can follow specific tutorials that meet your specific needs. For more information, see [Using AWS Lambda with Other Services \(p. 132\)](#).

The following topics provide additional information about AWS Lambda:

- [AWS Lambda Function Versioning and Aliases \(p. 47\)](#)
- [Using Amazon CloudWatch \(p. 226\)](#)
- [Best Practices for Working with AWS Lambda Functions \(p. 127\)](#)
- [AWS Lambda Limits \(p. 7\)](#)

Getting Started with AWS Lambda

To get started with AWS Lambda, use the Lambda console to create a function. In a few minutes, you can create a function, invoke it, and view logs, metrics, and trace data.

To use Lambda and other AWS services, you need an AWS account. If you don't have an account, visit aws.amazon.com and choose **Create an AWS Account**. For detailed instructions, see [Create and Activate an AWS Account](#).

As a best practice, you should also create an AWS Identity and Access Management (IAM) user with administrator permissions and use that for all work that does not require root credentials. Create a password for console access, and access keys to use command line tools. See [Creating Your First IAM Admin User and Group](#) in the *IAM User Guide* for instructions.

Sections

- [Create a Lambda Function with the Console \(p. 3\)](#)
- [AWS Lambda Concepts \(p. 5\)](#)
- [Command Line Tools \(p. 6\)](#)
- [AWS Lambda Limits \(p. 7\)](#)

Create a Lambda Function with the Console

In this Getting Started exercise you create a Lambda function using the AWS Lambda console. Next, you manually invoke the Lambda function using sample event data. AWS Lambda executes the Lambda function and returns results. You then verify execution results, including the logs that your Lambda function created and various CloudWatch metrics.

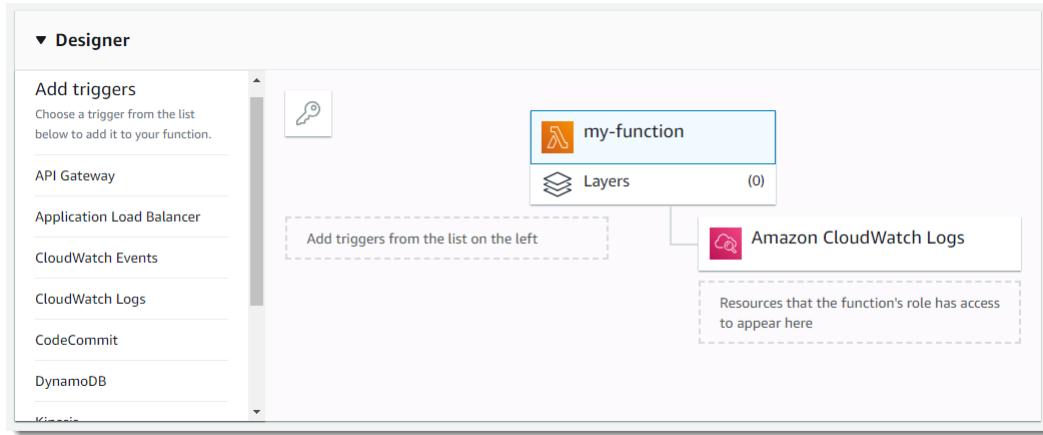
To create a Lambda function

1. Open the [AWS Lambda console](#).
2. Choose **Create a function**.
3. For **Function name**, enter **my-function**.
4. Choose **Create function**.

Lambda creates a Node.js function and an execution role that grants the function permission to upload logs. Lambda assumes the execution role when you invoke your function, and uses it to create credentials for the AWS SDK and to read data from event sources.

Use the Designer

The **Designer** lets you configure triggers and view permissions.



Choose **Amazon CloudWatch Logs** to view the log-related permissions that the execution role grants the function. When you add a trigger or configure features that require additional permissions, Lambda modifies to the function's execution role or resource-based policy to grant the minimum required access. To view these policies, choose the key icon.

Choose **my-function** in the designer to return to the function's code and configuration. For scripting languages, Lambda includes sample code that returns a success response. You can edit your function code with the embedded [AWS Cloud9](#) editor as long as your source code doesn't exceed the 3 MB limit.

Invoke the Lambda Function

Invoke your Lambda function using the sample event data provided in the console.

To invoke a function

1. In the upper right corner, choose **Test**.
2. In the **Configure test event** page, choose **Create new test event** and in **Event template**, leave the default **Hello World** option. Enter an **Event name** and note the following sample event template:

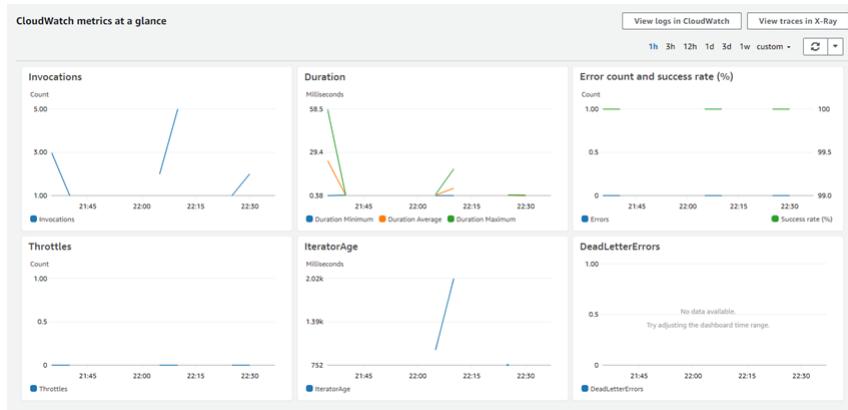
```
{
  "key3": "value3",
  "key2": "value2",
  "key1": "value1"
}
```

You can change key and values in the sample JSON, but don't change the event structure. If you do change any keys and values, you must update the sample code accordingly.

3. Choose **Create** and then choose **Test**. Each user can create up to 10 test events per function. Those test events are not available to other users.
4. AWS Lambda executes your function on your behalf. The `handler` in your Lambda function receives and then processes the sample event.
5. Upon successful execution, view results in the console.
 - The **Execution result** section shows the execution status as **succeeded** and also shows the function execution results, returned by the `return` statement.
 - The **Summary** section shows the key information reported in the **Log output** section (the *REPORT* line in the execution log).
 - The **Log output** section shows the log AWS Lambda generates for each execution. These are the logs written to CloudWatch by the Lambda function. The AWS Lambda console shows these logs for your convenience.

Note that the **Click here** link shows logs in the CloudWatch console. The function then adds logs to Amazon CloudWatch in the log group that corresponds to the Lambda function.

6. Run the Lambda function a few times to gather some metrics that you can view in the next step.
7. Choose **Monitoring**. This page shows graphs for the metrics that Lambda sends to CloudWatch.



For more information on these graphs, see [Accessing Amazon CloudWatch Metrics for AWS Lambda \(p. 228\)](#).

AWS Lambda Concepts

AWS Lambda lets you run *functions* in a serverless environment to process events in the language of your choice. Each instance of your function runs in an isolated execution context and processes one event at a time. When it finishes processing the event, it returns a response and Lambda sends it another event. Lambda automatically scales up the number of instances of your function to handle high numbers of events.

- **Function** – A script or program that runs in AWS Lambda. Lambda passes invocation events to your function. The function processes an event and returns a response. For more information, see [Working with Lambda Functions \(p. 22\)](#).
- **Runtimes** – Lambda runtimes allow functions in different languages to run in the same base execution environment. You configure your function to use a runtime that matches your programming language. The runtime sits in-between the Lambda service and your function code, relaying invocation events, context information, and responses between the two. You can use runtimes provided by Lambda, or build your own. For more information, see [AWS Lambda Runtimes \(p. 102\)](#).
- **Layers** – Lambda layers are a distribution mechanism for libraries, custom runtimes, and other function dependencies. Layers let you manage your in-development function code independently from the unchanging code and resources that it uses. You can configure your function to use layers that you create, layers provided by AWS, or layers from other AWS customers. For more information, see [AWS Lambda Layers \(p. 64\)](#).
- **Event source** – An AWS service, such as Amazon SNS, or a custom service, that triggers your function and executes its logic. For more information, see [AWS Lambda Event Source Mapping \(p. 82\)](#).
- **Downstream resources** – An AWS service, such as DynamoDB tables or Amazon S3 buckets, that your Lambda function calls once it is triggered.
- **Log streams** – While Lambda automatically monitors your function invocations and reports metrics to CloudWatch, you can annotate your function code with custom logging statements that allow you to analyze the execution flow and performance of your Lambda function to ensure it's working properly.

- **AWS SAM** – A model to define [serverless applications](#). AWS SAM is natively supported by AWS CloudFormation and defines simplified syntax for expressing serverless resources. For more information, see [What Is AWS SAM?](#) in the [AWS Serverless Application Model Developer Guide](#).

Command Line Tools

Install the AWS Command Line Interface to manage and use Lambda functions from the command line. Tutorials in this guide use the AWS CLI, which has commands for all Lambda API actions. Some functionality is not available in the Lambda console and can only be accessed with the AWS CLI or the AWS SDK.

The AWS SAM CLI is a separate command line tool that you can use to manage and test AWS SAM applications. In addition to commands for uploading artifacts and launching AWS CloudFormation stacks that are also available in the AWS CLI, the SAM CLI provides additional commands for validating templates and running applications locally in a Docker container.

Set Up the AWS CLI

To set up the AWS CLI

1. Download and configure the AWS CLI. For instructions, see the following topics in the [AWS Command Line Interface User Guide](#).
 - [Getting Set Up with the AWS Command Line Interface](#)
 - [Configuring the AWS Command Line Interface](#)
2. Add a named profile for the administrator user in the [AWS CLI config file](#). You use this profile when executing the AWS CLI commands. For more information on creating this profile, see [Named Profiles](#).

```
[profile adminuser]
aws_access_key_id = adminuser access key ID
aws_secret_access_key = adminuser secret access key
region = aws-region
```

For a list of available AWS regions, see [Regions and Endpoints](#) in the [Amazon Web Services General Reference](#).

3. Verify the setup by entering the following commands at the command prompt.
 - Try the help command to verify that the AWS CLI is installed on your computer:

```
$ aws help
```

- Try a Lambda command to verify the user can reach AWS Lambda. This command lists Lambda functions in the account, if any. The AWS CLI uses the `adminuser` credentials to authenticate the request.

```
$ aws lambda list-functions
```

AWS Serverless Application Model CLI

The AWS SAM CLI is a command line tool that operates on an AWS SAM template and application code. With the AWS SAM CLI, you can invoke Lambda functions locally, create a deployment package for your serverless application, deploy your serverless application to the AWS Cloud, and so on.

For more details about installing the AWS SAM CLI, see [Installing the AWS SAM CLI in the AWS Serverless Application Model Developer Guide](#).

AWS Lambda Limits

AWS Lambda limits the amount of compute and storage resources that you can use to run and store functions. The following limits apply per-region and can be increased. To request an increase, use the [Support Center console](#).

Resource	Default Limit
Concurrent executions (p. 37)	1,000
Function and layer storage	75 GB

For details on how Lambda scales your function concurrency in response to traffic, see [Understanding Scaling Behavior \(p. 86\)](#).

The following limits apply to function configuration, deployments, and execution. They cannot be changed.

Resource	Limit
Function memory allocation (p. 35)	128 MB to 3,008 MB, in 64 MB increments.
Function timeout (p. 35)	900 seconds (15 minutes)
Function environment variables (p. 40)	4 KB
Function resource-based policy (p. 10)	20 KB
Function layers (p. 64)	5 layers
Invocation frequency (requests per second)	10x concurrent executions limit (synchronous (p. 82) – all sources) 10x concurrent executions limit (asynchronous – non-AWS sources) Unlimited (asynchronous – AWS service sources (p. 132))
Invocation payload (p. 80) (request and response)	6 MB (synchronous) 256 KB (asynchronous)
Deployment package (p. 32) size	50 MB (zipped, for direct upload) 250 MB (unzipped, including layers) 3 MB (console editor)
Test events (console editor)	10
/tmp directory storage	512 MB

Resource	Limit
File descriptors	1,024
Execution processes/threads	1,024

Limits for other services, such as AWS Identity and Access Management, Amazon CloudFront (Lambda@Edge), and Amazon Virtual Private Cloud, can impact your Lambda functions. For more information, see [AWS Service Limits](#) and [Using AWS Lambda with Other Services \(p. 132\)](#).

AWS Lambda Permissions

You can use AWS Identity and Access Management (IAM) to manage access to the Lambda API and resources like functions and layers. For users and applications in your account that use Lambda, you manage permissions in a permissions policy that you can apply to IAM users, groups, or roles. To grant permissions to other accounts or AWS services that use your Lambda resources, you use a policy that applies to the resource itself.

A Lambda function also has a policy, called an [execution role \(p. 9\)](#), that grants it permission to access AWS services and resources. At a minimum, your function needs access to Amazon CloudWatch Logs for log streaming. If you [use AWS X-Ray to trace your function \(p. 232\)](#), or your function accesses services with the AWS SDK, you grant it permission to call them in the execution role. Lambda also uses the execution role to get permission to read from event sources when you use an [event source mapping \(p. 82\)](#) to trigger your function.

Note

If your function needs network access to a resource like a relational database that isn't accessible through AWS APIs or the internet, [configure it to connect to your VPC \(p. 68\)](#).

Use [resource-based policies \(p. 10\)](#) to give other accounts and AWS services permission to use your Lambda resources. Lambda resources include functions, versions, aliases, and layer versions. Each of these resources has a permissions policy that applies when the resource is accessed, in addition to any policies that apply to the user. When an AWS service like Amazon S3 calls your Lambda function, the resource-based policy gives it access.

To manage permissions for users and applications in your accounts, [use the managed policies that Lambda provides \(p. 13\)](#), or write your own. The Lambda console uses multiple services to get information about your function's configuration and triggers. You can use the managed policies as-is, or as a starting point for more restrictive policies.

You can restrict user permissions by the resource an action affects and, in some cases, by additional conditions. For example, you can specify a pattern for the Amazon Resource Name (ARN) of a function that requires a user to include their user name in the name of functions that they create. Additionally, you can add a condition that requires that the user configure functions to use a specific layer to, for example, pull in logging software. For the resources and conditions that are supported by each action, see [Resources and Conditions \(p. 18\)](#).

For more information about IAM, see [What Is IAM?](#) in the *IAM User Guide*.

Topics

- [AWS Lambda Execution Role \(p. 9\)](#)
- [Using Resource-based Policies for AWS Lambda \(p. 10\)](#)
- [Identity-based IAM Policies for AWS Lambda \(p. 13\)](#)
- [Resources and Conditions for Lambda Actions \(p. 18\)](#)

AWS Lambda Execution Role

An AWS Lambda function's execution role grants it permission to access AWS services and resources. You provide this role when you create a function, and Lambda assumes the role when your function is invoked. You can create an execution role for development that has permission to send logs to Amazon CloudWatch, and upload trace data to AWS X-Ray.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties:
 - **Trusted entity – AWS Lambda**
 - **Permissions – AWSLambdaBasicExecutionRole, AWSXrayWriteOnlyAccess**
 - **Role name – lambda-role**

You can add or remove permissions from a function's execution role at any time, or configure your function to use a different role. Add permissions for any services that your function calls with the AWS SDK, and for services that Lambda uses to enable optional features.

The following managed policies provide permissions that are required to use Lambda features:

- **AWSLambdaBasicExecutionRole** – Permission to upload logs to CloudWatch.
- **AWSLambdaKinesisExecutionRole** – Permission to read events from an Amazon Kinesis data stream or consumer.
- **AWSLambdaDynamoDBExecutionRole** – Permission to read records from an Amazon DynamoDB stream.
- **AWSLambdaSQSQueueExecutionRole** – Permission to read a message from an Amazon Simple Queue Service (Amazon SQS) queue.
- **AWSLambdaVPCAccessExecutionRole** – Permission to manage elastic network interfaces to connect your function to a VPC.
- **AWSXrayWriteOnlyAccess** – Permission to upload trace data to X-Ray.

When you use an [event source mapping \(p. 82\)](#) to invoke your function, Lambda uses the execution role to read event data. For example, an event source mapping for Amazon Kinesis reads events from a data stream and sends them to your function in batches. You can use event source mappings with the following services:

Services That Lambda Reads Events From

- [Amazon Kinesis \(p. 180\)](#)
- [Amazon DynamoDB \(p. 170\)](#)
- [Amazon Simple Queue Service \(p. 216\)](#)

In addition to the managed policies, the Lambda console provides templates for creating a custom policy that has the permissions related to additional use cases. When you create a function, you can choose to create a new execution role with permissions from one or more templates. These templates are also applied automatically when you create a function from a blueprint, or when you configure options that require access to other services. Example templates are available in this guide's [GitHub repository](#).

Using Resource-based Policies for AWS Lambda

AWS Lambda supports resource-based permissions policies for Lambda functions and layers. Resource-based policies let you grant usage permission to other accounts on a per-resource basis. You also use a resource-based policy to allow an AWS service to invoke your function.

For Lambda functions, you can [grant an account permission \(p. 12\)](#) to invoke or manage a function. You can add multiple statements to grant access to multiple accounts, or let any account invoke your

function. For functions that another AWS service invokes in response to activity in your account, you use the policy to [grant invoke permission to the service \(p. 11\)](#).

For Lambda layers, you use a resource-based policy on a version of the layer to let other accounts use it. In addition to policies that grant permission to a single account or all accounts, for layers, you can also grant permission to all accounts in an organization.

Note

You can only update resource-based policies for Lambda resources within the scope of the [AddPermission \(p. 343\)](#) and [AddLayerVersionPermission \(p. 339\)](#) API actions. You can't author policies for your Lambda resources in JSON, or use conditions that don't map to parameters for those actions.

Resource-based policies apply to a single function, version, alias, or layer version. They grant permission to one or more services and accounts. For trusted accounts that you want to have access to multiple resources, or to use API actions that resource-based policies don't support, you can use [cross-account roles \(p. 13\)](#).

Topics

- [Granting Function Access to AWS Services \(p. 11\)](#)
- [Granting Function Access to Other Accounts \(p. 12\)](#)
- [Granting Layer Access to Other Accounts \(p. 12\)](#)
- [Cleaning up Resource-based Policies \(p. 13\)](#)

Granting Function Access to AWS Services

When you [use an AWS service to invoke your function \(p. 83\)](#), you grant permission in a statement on a resource-based policy. You can apply the statement to the function, or limit it to a single version or alias.

Note

When you add a trigger to your function with the Lambda console, the console updates the function's resource-based policy to allow the service to invoke it. To grant permissions to other accounts or services that aren't available in the Lambda console, use the AWS CLI.

Add a statement with the `add-permission` command. The simplest resource-based policy statement allows a service to invoke a function. The following command grants Amazon SNS permission to invoke a function named `my-function`.

```
$ aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --statement-id sns \
--principal sns.amazonaws.com --output text
{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:us-east-2:123456789012:function:my-function"}
```

This lets Amazon SNS invoke the function, but it doesn't restrict the Amazon SNS topic that triggers the invocation. To ensure that your function is only invoked by a specific resource, specify the Amazon Resource Name (ARN) of the resource with the `source-arn` option. The following command only allows Amazon SNS to invoke the function for subscriptions to a topic named `my-topic`.

```
$ aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --statement-id sns-my-topic \
--principal sns.amazonaws.com --source-arn arn:aws:sns:us-east-2:123456789012:my-topic
```

Some services can invoke functions in other accounts. If you specify a source ARN that has your account ID in it, that isn't an issue. For Amazon S3, however, the source is a bucket whose ARN doesn't have an

account ID in it. It's possible that you could delete the bucket and another account could create a bucket with the same name. Use the account-id option to ensure that only resources in your account can invoke the function.

```
$ aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --statement-id s3-account \
--principal s3.amazonaws.com --source-arn arn:aws:s3:::my-bucket-123456 --source-account 123456789012
```

Granting Function Access to Other Accounts

To grant permissions to another AWS account, specify the account ID as the principal. The following example grants account 210987654321 permission to invoke my-function with the prod alias.

```
$ aws lambda add-permission --function-name my-function:prod --statement-id xaccount --action lambda:InvokeFunction \
--principal 210987654321 --output text
{"Sid": "xaccount", "Effect": "Allow", "Principal": {"AWS": "arn:aws:iam::210987654321:root"}, "Action": "lambda:InvokeFunction", "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function"}
```

The [alias \(p. 51\)](#) limits which version the other account can invoke. It requires the other account to include the alias in the function ARN.

```
$ aws lambda invoke --function-name arn:aws:lambda:us-west-2:123456789012:function:my-function:prod out
{
    "StatusCode": 200,
    "ExecutedVersion": "1"
}
```

You can then update the alias to point to new versions as needed. When you update the alias, the other account doesn't need to change its code to use the new version, and it only has permission to invoke the version that you choose.

You can grant cross-account access for any API action that [operates on an existing function \(p. 19\)](#). For example, you could grant access to lambda>ListAliases to let an account get a list of aliases, or lambda>GetFunction to let them download your function code. Add each permission separately, or use lambda:* to grant access to all actions for the specified function.

To grant other accounts permission for multiple functions, or for actions that don't operate on a function, use [roles \(p. 13\)](#).

Granting Layer Access to Other Accounts

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

```
$ aws lambda add-layer-version-permission --layer-name xray-sdk-nodejs --statement-id xaccount \
--action lambda:GetLayerVersion --principal 210987654321 --version-number 1 --output text
e210ffdc-e901-43b0-824b-5fc0dd26d16 {"Sid": "xaccount", "Effect": "Allow", "Principal": {"AWS": "arn:aws:iam::210987654321:root"}, "Action": "lambda:GetLayerVersion", "Resource": "arn:aws:lambda:us-east-2:123456789012:layer:xray-sdk-nodejs:1"}
```

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

To grant permission to all accounts in an organization, use the `organization-id` option. The following example grants all accounts in an organization permission to use version 3 of a layer.

```
$ aws lambda add-layer-version-permission --layer-name my-layer \
--statement-id engineering-org --version-number 3 --principal '*' \
--action lambda:GetLayerVersion --organization-id o-t194hfs8cz --output text
b0cd9796-d4eb-4564-939f-de7fe0b42236 {"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:lambda:us-
east-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":{"aws:PrincipalOrgID":"o-
t194hfs8cz"}}}"
```

To grant permission to all AWS accounts, use `*` for the principal, and omit the organization ID. For multiple accounts or organizations, add multiple statements.

Cleaning up Resource-based Policies

To view a function's resource-based policy, use the `get-policy` command.

```
$ aws lambda get-policy --function-name my-function --output text
{"Version":"2012-10-17","Id":"default","Statement":
[{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"s3.amazonaws.com"}, "Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:us-
east-2:123456789012:function:my-function","Condition":{"ArnLike":
{"AWS:SourceArn":"arn:aws:sns:us-east-2:123456789012:lambda*"}}}]}      7c681fc9-b791-4e91-
acdf-eb847fd8aa0f0
```

For versions and aliases, append the version number or alias to the function name.

```
$ aws lambda get-policy --function-name my-function:PROD
```

To remove permissions from your function, use `remove-permission`.

```
$ aws lambda remove-permission --function-name example --statement-id sns
```

Use the `get-layer-version-policy` command to view the permissions on a layer, and `remove-layer-version-permission` to remove statements from the policy.

```
$ aws lambda get-layer-version-policy --layer-name my-layer --version-number 3 --output
text
b0cd9796-d4eb-4564-939f-de7fe0b42236 {"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:lambda:us-
west-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":{"aws:PrincipalOrgID":"o-
t194hfs8cz"}}}"
```



```
$ aws lambda remove-layer-version-permission --layer-name my-layer --version-number 3 --
statement-id engineering-org
```

Identity-based IAM Policies for AWS Lambda

You can use identity-based policies in AWS Identity and Access Management (IAM) to grant users in your account access to Lambda. Identity-based policies can apply to users directly, or to groups and roles that are associated with a user. You can also grant users in another account permission to assume a role in your account and access your Lambda resources.

Lambda provides managed policies that grant access to Lambda API actions and, in some cases, access to other services used to develop and manage Lambda resources. Lambda updates the managed policies as needed, to ensure that your users have access to new features when they're released.

- **AWSLambdaFullAccess** – Grants full access to AWS Lambda actions and other services used to develop and maintain Lambda resources.
- **AWSLambdaReadOnlyAccess** – Grants read-only access to AWS Lambda resources.
- **AWSLambdaRole** – Grants permissions to invoke Lambda functions.

Managed policies grant permission to API actions without restricting the functions or layers that a user can modify. For finer-grained control, you can create your own policies that limit the scope of a user's permissions.

Sections

- [Function Development \(p. 14\)](#)
- [Layer Development and Use \(p. 17\)](#)
- [Cross-Account Roles \(p. 18\)](#)

Function Development

The following shows an example of a permissions policy with limited scope. It allows a user to create and manage Lambda functions named with a designated prefix (`intern-`), and configured with a designated execution role.

Example Function Development Policy

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "ReadOnlyPermissions",  
            "Effect": "Allow",  
            "Action": [  
                "lambda:GetAccountSettings",  
                "lambda>ListFunctions",  
                "lambda>ListTags",  
                "lambda:GetEventSourceMapping",  
                "lambda>ListEventSourceMappings",  
                "iam>ListRoles"  
            ],  
            "Resource": "*"  
        },  
        {  
            "Sid": "DevelopFunctions",  
            "Effect": "Allow",  
            "NotAction": [  
                "lambda:AddPermission",  
                "lambda:PutFunctionConcurrency"  
            ],  
            "Resource": "arn:aws:lambda:*::function:intern-*"  
        },  
        {  
            "Sid": "DevelopEventSourceMappings",  
            "Effect": "Allow",  
            "Action": [  
                "lambda>DeleteEventSourceMapping",  
                "lambda:UpdateEventSourceMapping",  
                "lambda>CreateEventSourceMapping"  
            ]  
        }  
    ]  
}
```

```

        ],
        "Resource": "*",
        "Condition": {
            "StringLike": {
                "lambda:FunctionArn": "arn:aws:lambda:*::function:intern-*"
            }
        }
    },
    {
        "Sid": "PassExecutionRole",
        "Effect": "Allow",
        "Action": [
            "iam>ListRolePolicies",
            "iam>ListAttachedRolePolicies",
            "iam:GetRole",
            "iam:PassRole"
        ],
        "Resource": "arn:aws:iam::role/intern-lambda-execution-role"
    },
    {
        "Sid": "ViewExecutionRolePolicies",
        "Effect": "Allow",
        "Action": [
            "iam:GetPolicy",
            "iam:GetPolicyVersion"
        ],
        "Resource": "arn:aws:iam::aws:policy/*"
    },
    {
        "Sid": "ViewLogs",
        "Effect": "Allow",
        "Action": [
            "logs:)"
        ],
        "Resource": "arn:aws:logs::log-group:/aws/lambda/intern-*"
    }
]
}

```

The permissions in the policy are organized into statements based on the [resources and conditions \(p. 18\)](#) that they support.

- **ReadOnlyPermissions** – The Lambda console uses these permissions when you browse and view functions. They don't support resource patterns or conditions.

```

        "Action": [
            "lambda:GetAccountSettings",
            "lambda>ListFunctions",
            "lambda>ListTags",
            "lambda:GetEventSourceMapping",
            "lambda>ListEventSourceMappings",
            "iam>ListRoles"
        ],
        "Resource": "*"

```

- **DevelopFunctions** – Use any Lambda action that operates on functions prefixed with *intern-*, *except AddPermission and PutFunctionConcurrency*. **AddPermission** modifies the [resource-based policy \(p. 10\)](#) on the function and can have security implications. **PutFunctionConcurrency** reserves scaling capacity for a function and can take capacity away from other functions.

```

    "NotAction": [
        "lambda:AddPermission",
        "lambda:PutFunctionConcurrency"
    ],
    "Resource": "arn:aws:lambda:*:*:function:intern-*"

```

- **DevelopEventSourceMappings** – Manage event source mappings on functions that are prefixed with `intern-`. These actions operate on event source mappings, but you can restrict them by function with a *condition*.

```

    "Action": [
        "lambda>DeleteEventSourceMapping",
        "lambda>UpdateEventSourceMapping",
        "lambda>CreateEventSourceMapping"
    ],
    "Resource": "*",
    "Condition": {
        "StringLike": {
            "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
        }
    }
}

```

- **PassExecutionRole** – View and pass only a role named `intern-lambda-execution-role`, which must be created and managed by a user with IAM permissions. `PassRole` is used when you assign an execution role to a function.

```

    "Action": [
        "iam>ListRolePolicies",
        "iam>ListAttachedRolePolicies",
        "iam:GetRole",
        "iam:PassRole"
    ],
    "Resource": "arn:aws:iam:::role/intern-lambda-execution-role"

```

- **ViewExecutionRolePolicies** – View the AWS-provided managed policies that are attached to the execution role. This lets you view the function's permissions in the console, but doesn't include permission to view policies that were created by other users in the account.

```

    "Action": [
        "iam:GetPolicy",
        "iam:GetPolicyVersion"
    ],
    "Resource": "arn:aws:iam::aws:policy/*"

```

- **ViewLogs** – Use CloudWatch Logs to view logs for functions that are prefixed with `intern-`.

```

    "Action": [
        "logs:*log"
    ],
    "Resource": "arn:aws:logs::*:log-group:/aws/lambda/intern-*"

```

This policy allows a user to get started with Lambda, without putting other users' resources at risk. It doesn't allow a user to configure a function to be triggered by or call other AWS services, which requires broader IAM permissions. It also doesn't include permission to services that don't support limited-scope policies, like CloudWatch and X-Ray. Use the read-only policies for these services to give the user access to metrics and trace data.

When you configure triggers for your function, you need access to use the AWS service that invokes your function. For example, to configure an Amazon S3 trigger, you need permission to Amazon S3 actions to manage bucket notifications. Many of these permissions are included in the **AWSLambdaFullAccess** managed policy. Example policies are available in this guide's [GitHub repository](#).

Layer Development and Use

The following policy grants a user permission to create layers and use them with functions. The resource patterns allow the user to work in any AWS Region and with any layer version, as long as the name of the layer starts with `test-`.

Example Layer Development Policy

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "PublishLayers",  
            "Effect": "Allow",  
            "Action": [  
                "lambda:PublishLayerVersion"  
            ],  
            "Resource": "arn:aws:lambda:*::layer:test-*"  
        },  
        {  
            "Sid": "ManageLayerVersions",  
            "Effect": "Allow",  
            "Action": [  
                "lambda:GetLayerVersion",  
                "lambda:DeleteLayerVersion"  
            ],  
            "Resource": "arn:aws:lambda:*::layer:test-*::*"  
        }  
    ]  
}
```

You can also enforce layer use during function creation and configuration with the `lambda:Layer` condition. For example, you can prevent users from using layers published by other accounts. The following policy adds a condition to the `CreateFunction` and `UpdateFunctionConfiguration` actions to require that any layers specified come from account 123456789012.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "ConfigureFunctions",  
            "Effect": "Allow",  
            "Action": [  
                "lambda>CreateFunction",  
                "lambda:UpdateFunctionConfiguration"  
            ],  
            "Resource": "*",  
            "Condition": {  
                "ForAllValues:StringLike": {  
                    "lambda:Layer": [  
                        "arn:aws:lambda:*:123456789012:layer::*"  
                    ]  
                }  
            }  
        }  
    ]  
}
```

}

To ensure that the condition applies, verify that no other statements grant the user permission to these actions.

Cross-Account Roles

You can apply any of the preceding policies and statements to a role, which you can then share with another account to give it access to your Lambda resources. Unlike an IAM user, a role doesn't have credentials for authentication. Instead, it has a *trust policy* that specifies who can assume the role and use its permissions.

You can use cross-account roles to give accounts that you trust access to Lambda actions and resources. If you just want to grant permission to invoke a function or use a layer, use [resource-based policies \(p. 10\)](#) instead.

For more information, see [IAM Roles](#) in the *IAM User Guide*.

Resources and Conditions for Lambda Actions

You can restrict the scope of a user's permissions by specifying resources and conditions in an IAM policy. Each API action supports a combination of resource and condition types that varies depending on the behavior of the action.

Every IAM policy statement grants permission to an action that's performed on a resource. When the action doesn't act on a named resource, or when you grant permission to perform the action on all resources, the value of the resource in the policy is a wildcard (*). For many API actions, you can restrict the resources that a user can modify by specifying the Amazon Resource Name (ARN) of a resource, or an ARN pattern that matches multiple resources.

To restrict permissions by resource, specify the resource by ARN.

Lambda Resource ARN Format

- Function – arn:aws:lambda:**us-west-2:123456789012:function:my-function**
- Function version – arn:aws:lambda:**us-west-2:123456789012:function:my-function:1**
- Function alias – arn:aws:lambda:**us-west-2:123456789012:function:my-function:TEST**
- Event source mapping – arn:aws:lambda:**us-west-2:123456789012:event-source-mapping:fa123456-14a1-4fd2-9fec-83de64ad683de6d47**
- Layer – arn:aws:lambda:**us-west-2:123456789012:layer:my-layer**
- Layer version – arn:aws:lambda:**us-west-2:123456789012:layer:my-layer:1**

For example, the following policy allows a user in account 123456789012 to invoke a function named my-function in the US West (Oregon) Region.

Example Invoke a Function Policy

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "Invoke",  
            "Effect": "Allow",  
            "Action": [
```

```
        "lambda:InvokeFunction"
    ],
    "Resource": "arn:aws:lambda:us-west-2:123456789012:function:my-function"
}
]
```

This is a special case where the action identifier (`lambda:InvokeFunction`) differs from the API operation ([Invoke \(p. 400\)](#)). For other actions, the action identifier is the operation name prefixed by `lambda:`.

Conditions are an optional policy element that applies additional logic to determine if an action is allowed. In addition to [common conditions](#) supported by all actions, Lambda defines condition types that you can use to restrict the values of additional parameters on some actions.

For example, the `lambda:Principal` condition lets you restrict the service or account that a user can grant invocation access to on a function's resource-based policy. The following policy lets a user grant permission to SNS topics to invoke a function named `test`.

Example Manage Function Policy Permissions

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ManageFunctionPolicy",
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:test:*",
            "Condition": {
                "StringEquals": {
                    "lambda:Principal": "sns.amazonaws.com"
                }
            }
        }
    ]
}
```

The condition requires that the principal is Amazon SNS and not another service or account. The resource pattern requires that the function name is `test` and includes a version number or alias. For example, `test:v1`.

For more information on resources and conditions for Lambda and other AWS services, see [Actions, Resources, and Condition Keys](#) in the *IAM User Guide*.

Sections

- [Functions \(p. 19\)](#)
- [Event Source Mappings \(p. 21\)](#)
- [Layers \(p. 21\)](#)

Functions

Actions that operate on a function can be restricted to a specific function by function, version, or alias ARN, as described in the following table. Actions that don't support resource restrictions can only be granted for all resources (*).

Functions

Action	Resource	Condition
AddPermission (p. 343)	Function	<code>lambda:Principal</code>
RemovePermission (p. 441)	Function version Function alias	
Invoke (p. 400) Permission: <code>lambda:InvokeFunction</code>	Function Function version Function alias	None
CreateFunction (p. 355) UpdateFunctionConfiguration (p. 463)	Function	<code>lambda:Layer</code>
CreateAlias (p. 347) DeleteAlias (p. 363) DeleteFunction (p. 368) DeleteFunctionConcurrency (p. 370) GetAlias (p. 376) GetFunction (p. 382) GetFunctionConfiguration (p. 385) GetPolicy (p. 398) ListAliases (p. 407) ListVersionsByFunction (p. 423) PublishVersion (p. 430) PutFunctionConcurrency (p. 436) UpdateAlias (p. 448) UpdateFunctionCode (p. 456)	Function	None
GetAccountSettings (p. 374) ListFunctions (p. 413) ListTags (p. 421) TagResource (p. 444) UntagResource (p. 446)	*	None

Event Source Mappings

For event source mappings, delete and update permissions can be restricted to a specific event source. The `lambda:FunctionArn` condition lets you restrict which functions a user can configure an event source to invoke.

For these actions, the resource is the event source mapping, so Lambda provides a condition that lets you restrict permission based on the function that the event source mapping invokes.

Event Source Mappings

Action	Resource	Condition
DeleteEventSourceMapping (p. 365)	Event source mapping	<code>lambda:FunctionArn</code>
UpdateEventSourceMapping (p. 452)		
CreateEventSourceMapping (p. 351)	*	<code>lambda:FunctionArn</code>
GetEventSourceMapping (p. 379)	*	None
ListEventSourceMappings (p. 410)		

Layers

Layer actions let you restrict the layers that a user can manage or use with a function. Actions related to layer use and permissions act on a version of a layer, while `PublishLayerVersion` acts on a layer name. You can use either with wildcards to restrict the layers that a user can work with by name.

Layers

Action	Resource	Condition
AddLayerVersionPermission (p. 339)	Layer version	None
RemoveLayerVersionPermission (p. 439)		
GetLayerVersion (p. 390)		
GetLayerVersionPolicy (p. 396)		
DeleteLayerVersion (p. 372)		
PublishLayerVersion (p. 426)	Layer	None
ListLayers (p. 416)	*	None
ListLayerVersions (p. 418)		

Working with Lambda Functions

If you are new to AWS Lambda, you might ask: How does AWS Lambda execute my code? How does AWS Lambda know the amount of memory and CPU requirements needed to run my Lambda code? The following sections provide an overview of how a Lambda function works.

In subsequent sections, we cover how the functions you create get invoked, and how to deploy and monitor them. We also recommend reading the **Function Code** and **Function Configuration** sections at [Best Practices for Working with AWS Lambda Functions \(p. 127\)](#).

To begin, we introduce you to the topic that explains the fundamentals of building a Lambda function, [Building Lambda Functions \(p. 22\)](#).

Topics

- [Building Lambda Functions \(p. 22\)](#)
- [Creating Functions Using the AWS Lambda Console Editor \(p. 24\)](#)
- [Programming Model \(p. 31\)](#)
- [Creating a Deployment Package \(p. 32\)](#)
- [Accessing AWS Resources from a Lambda Function \(p. 33\)](#)

Building Lambda Functions

You upload your application code in the form of one or more *Lambda functions* to AWS Lambda, a compute service. In turn, AWS Lambda executes the code on your behalf. AWS Lambda takes care of provisioning and managing the servers to run the code upon invocation.

Typically, the lifecycle for an AWS Lambda-based application includes authoring code, deploying code to AWS Lambda, and then monitoring and troubleshooting. The following are general questions that come up in each of these lifecycle phases:

- **Authoring code for your Lambda function** – What languages are supported? Is there a programming model that I need to follow? How do I package my code and dependencies for uploading to AWS Lambda? What tools are available?
- **Uploading code and creating Lambda functions** – How do I upload my code package to AWS Lambda? How do I tell AWS Lambda where to begin executing my code? How do I specify compute requirements like memory and timeout?
- **Monitoring and troubleshooting** – For my Lambda function that is in production, what metrics are available? If there are any failures, how do I get logs or troubleshoot issues?

The following sections provide introductory information and the Example section at the end provides working examples for you to explore.

Authoring Code for Your Lambda Function

You can author your Lambda function code in the languages that are supported by AWS Lambda. For a list of supported languages, see [AWS Lambda Runtimes \(p. 102\)](#). There are tools for authoring code, such as the AWS Lambda console, Eclipse IDE, and Visual Studio IDE. But the available tools and options depend on the following:

- Language you choose to write your Lambda function code.
- Libraries that you use in your code. AWS Lambda runtime provides some of the libraries and you must upload any additional libraries that you use.

The following table lists languages, and the available tools and options that you can use.

Language	Tools and Options for Authoring Code
Node.js	<ul style="list-style-type: none"> AWS Lambda console Visual Studio, with IDE plug-in (see AWS Lambda Support in Visual Studio) Your own authoring environment For more information, see Deploying Code and Creating a Lambda Function (p. 23).
Java	<ul style="list-style-type: none"> Eclipse, with AWS Toolkit for Eclipse (see Using AWS Lambda with the AWS Toolkit for Eclipse) Your own authoring environment For more information, see Deploying Code and Creating a Lambda Function (p. 23).
C#	<ul style="list-style-type: none"> Visual Studio, with IDE plug-in (see AWS Lambda Support in Visual Studio) .NET Core (see .NET Core installation guide) Your own authoring environment For more information, see Deploying Code and Creating a Lambda Function (p. 23).
Python	<ul style="list-style-type: none"> AWS Lambda console Your own authoring environment For more information, see Deploying Code and Creating a Lambda Function (p. 23).
Go	<ul style="list-style-type: none"> Your own authoring environment For more information, see Deploying Code and Creating a Lambda Function (p. 23).
PowerShell	<ul style="list-style-type: none"> Your own authoring environment PowerShell Core 6.0 (see Installing PowerShell Core) .NET Core 2.1 SDK (see .NET downloads) AWSLambdaPSCore Module (see PowerShell Gallery)

In addition, regardless of the language you choose, there is a pattern to writing Lambda function code. For example, how you write the handler method of your Lambda function (that is, the method that AWS Lambda first calls when it begins executing the code), how you pass events to the handler, what statements you can use in your code to generate logs in CloudWatch Logs, how to interact with AWS Lambda runtime and obtain information such as the time remaining before timeout, and how to handle exceptions. The [Programming Model \(p. 31\)](#) section provides information for each of the supported languages.

Deploying Code and Creating a Lambda Function

To create a Lambda function, you first package your code and dependencies in a deployment package. Then, you upload the deployment package to AWS Lambda to create your Lambda function.

Topics

- [Creating a Deployment Package \(p. 24\)](#)

- [Uploading a Deployment Package \(p. 24\)](#)
- [Testing a Lambda Function \(p. 24\)](#)

Creating a Deployment Package

You must first organize your code and dependencies in certain ways and create a *deployment package*. Instructions to create a deployment package vary depending on the language you choose to author the code. For example, you can use build plugins such as Jenkins (for Node.js and Python), and Maven (for Java) to create the deployment packages. For more information, see [Creating a Deployment Package \(p. 32\)](#).

When you create Lambda functions using the console, the console creates the deployment package for you, and then uploads it to create your Lambda function.

Uploading a Deployment Package

AWS Lambda provides the [CreateFunction \(p. 355\)](#) operation, which is what you use to create a Lambda function. You can use the AWS Lambda console, AWS CLI, and AWS SDKs to create a Lambda function. Internally, all of these interfaces call the `CreateFunction` operation.

In addition to providing your deployment package, you can provide configuration information when you create your Lambda function including the compute requirements of your Lambda function, the name of the handler method in your Lambda function, and the runtime, which depends on the language you chose to author your code. For more information, see [Working with Lambda Functions \(p. 22\)](#).

Testing a Lambda Function

If your Lambda function is designed to process events of a specific type, you can use sample event data to test your Lambda function using one of the following methods:

- Test your Lambda function in the console.
- Test your Lambda function using the AWS CLI. You can use the `Invoke` method to invoke your Lambda function and pass in sample event data.
- Test your Lambda function locally using the [AWS SAM CLI](#).

Monitoring and Troubleshooting

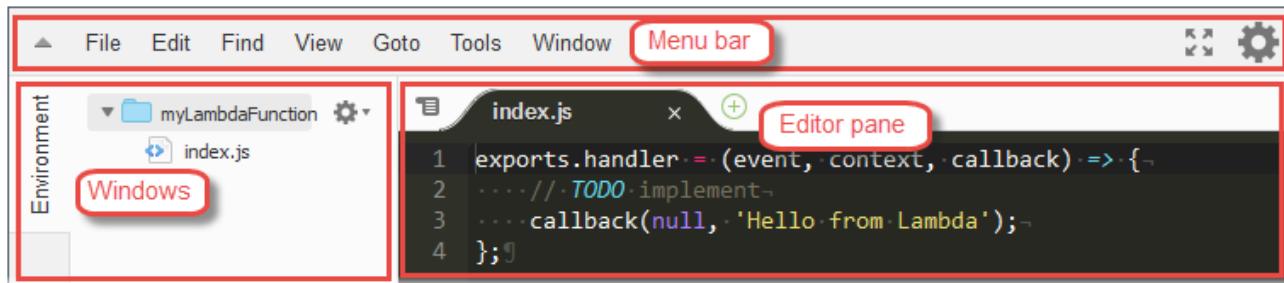
After your Lambda function is in production, AWS Lambda automatically monitors functions on your behalf, reporting metrics through Amazon CloudWatch. For more information, see [Accessing Amazon CloudWatch Metrics for AWS Lambda \(p. 228\)](#).

To help you troubleshoot failures in a function, Lambda logs all requests handled by your function and also automatically stores logs that your code generates in Amazon CloudWatch Logs. For more information, see [Accessing Amazon CloudWatch Logs for AWS Lambda \(p. 229\)](#).

Creating Functions Using the AWS Lambda Console Editor

The code editor in the AWS Lambda console enables you to write, test, and view the execution results of your Lambda function code.

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



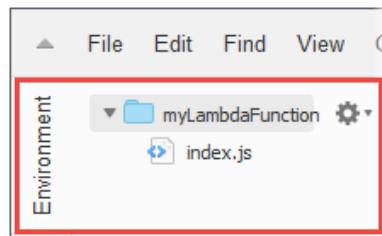
You use the menu bar to run common commands. For more information, see [Using the Menu Bar \(p. 29\)](#).

You use windows to work with files, folders, and other commands. For more information, see [Working with Files and Folders \(p. 25\)](#) and [Working with Commands \(p. 30\)](#).

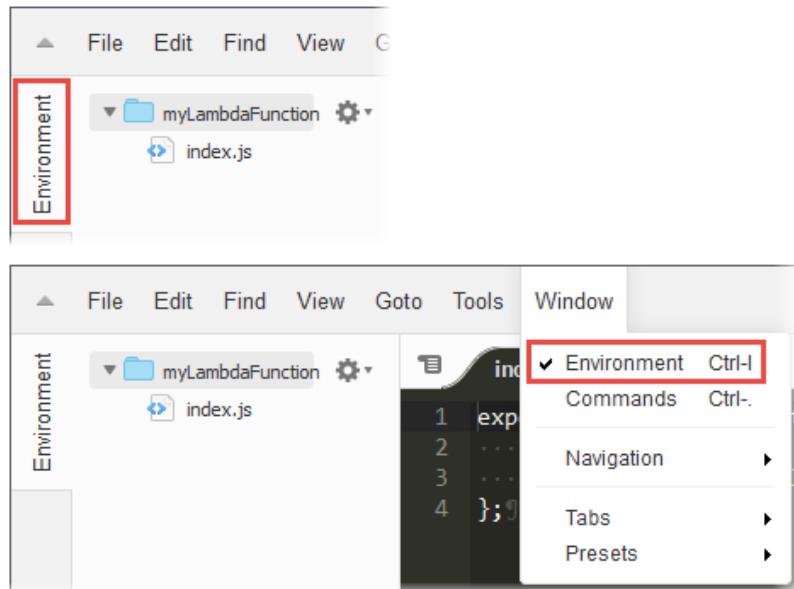
You use the editor pane to write code. For more information, see [Working with Code \(p. 27\)](#).

Working with Files and Folders

You can use the **Environment** window in the code editor to create, open, and manage files for your function.



To show or hide the **Environment** window, choose the **Environment** button. If the **Environment** button is not visible, choose **Window, Environment** on the menu bar.

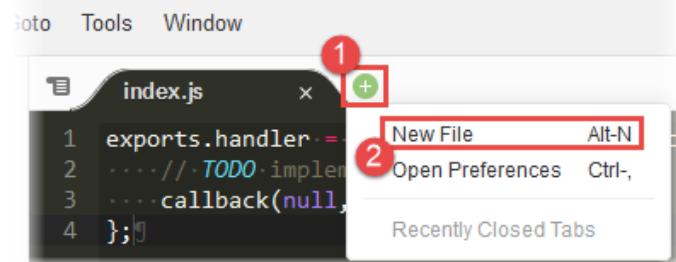


To open a single file and show its contents in the editor pane, double-click the file in the **Environment** window.

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

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

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



To create a new folder, right-click the folder in the **Environment** window where you want the new folder to go, and then choose **New Folder**. Type the folder's name, and then press **Enter**.

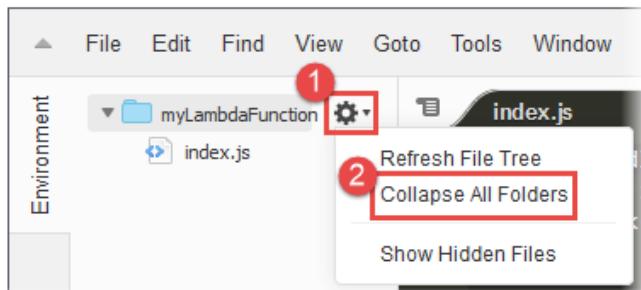
To save a file, with the file open and its contents visible in the editor pane, choose **File, Save** on the menu bar.

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

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

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

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

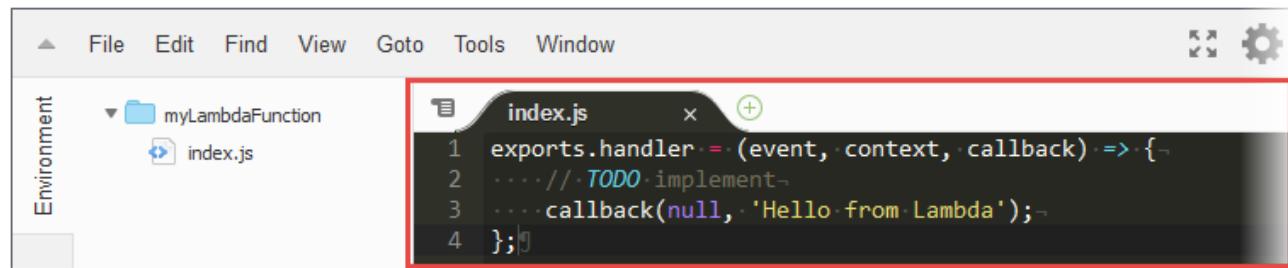


To show or hide hidden files, choose the gear icon in the **Environment** window, and then choose **Show Hidden Files**.

You can also create, open, and manage files by using the **Commands** window. For more information, see [Working with Commands \(p. 30\)](#).

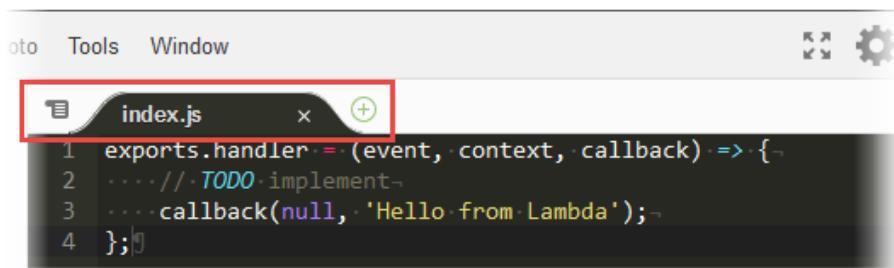
Working with Code

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



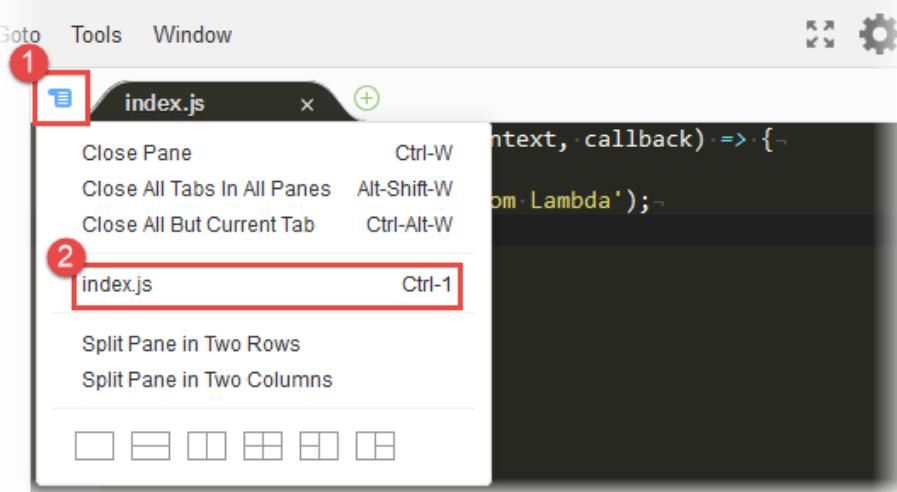
Working with Tab Buttons

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



To display an **open file's contents**, do one of the following:

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



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

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

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

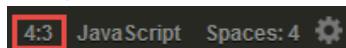
To create a new file, choose the + button in the tab buttons bar, and then choose **New File**. When you're ready to save the file, choose **File, Save** or **File, Save As** on the menu bar. Then use the **Save As** dialog box that displays to name the file and choose where to save it.

Working with the Status Bar

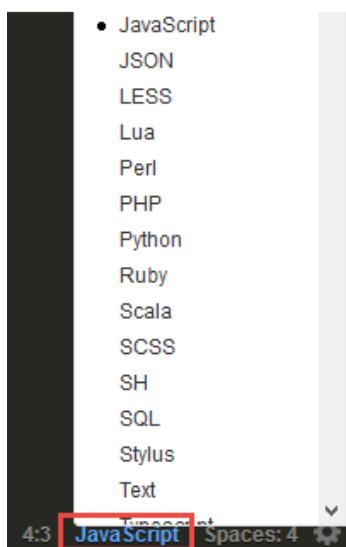
Use the status bar to move quickly to a line in the active file and to change how code is displayed.



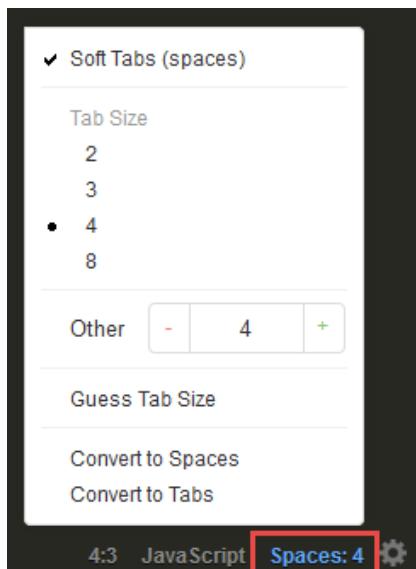
To move quickly to a line in the active file, choose the line selector, type the line number to go to, and then press **Enter**.



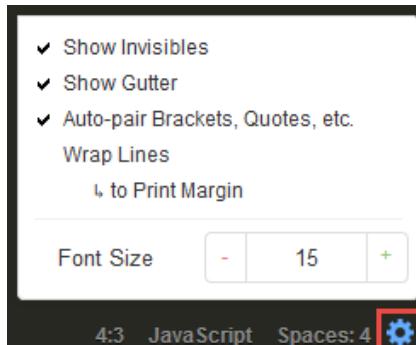
To change the code color scheme in the active file, choose the code color scheme selector, and then choose the new code color scheme.



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



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



Using the Menu Bar

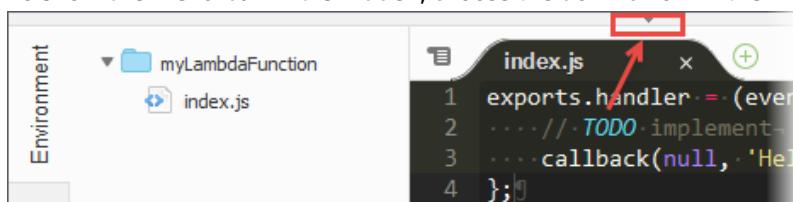
You can use the menu bar to run common commands.



To hide the menu bar, choose the up arrow in the menu bar.



To show the menu bar if it is hidden, choose the down arrow in the menu bar.



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

You can also run commands by using the **Commands** window. For more information, see [Working with Commands \(p. 30\)](#).

Working in Fullscreen Mode

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

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



To shrink the code editor to its original size, choose the **Toggle fullscreen** button again.

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

Working with Preferences

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

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



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

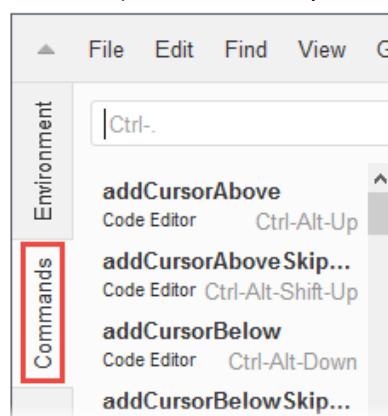
- [Project Setting Changes You Can Make](#)
- [User Setting Changes You Can Make](#)

Note that some of the settings listed in those references are not available in the code editor.

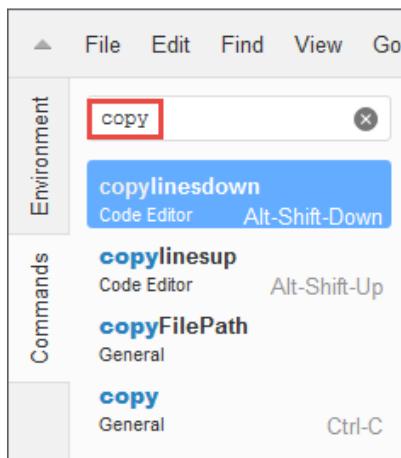
Working with Commands

You can use the **Commands** window to run various commands such as those found on the menu bar, in the **Environment** window, in the editor pane.

To show or hide the Commands window, choose the **Commands** button. If the **Commands** button is not visible, choose **Window, Commands** on the menu bar.



To run a command, choose it in the **Commands** window. To find a command, type some or all of the command's name in the search box.



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

Programming Model

You write code for your Lambda function in one of the languages AWS Lambda supports. Regardless of the language you choose, there is a common pattern to writing code for a Lambda function that includes the following core concepts:

- **Handler** – Handler is the function AWS Lambda calls to start execution of your Lambda function. You identify the handler when you create your Lambda function. When a Lambda function is invoked, AWS Lambda starts executing your code by calling the handler function. AWS Lambda passes any event data to this handler as the first parameter. Your handler should process the incoming event data and may invoke any other functions/methods in your code.
- **Context** – AWS Lambda also passes a `context` object to the handler function, as the second parameter. Via this context object your code can interact with AWS Lambda. For example, your code can find the execution time remaining before AWS Lambda terminates your Lambda function.

In addition, for languages such as Node.js, there is an asynchronous platform that uses callbacks. AWS Lambda provides additional methods on this context object. You use these context object methods to tell AWS Lambda to terminate your Lambda function and optionally return values to the caller.

- **Logging** – Your Lambda function can contain logging statements. AWS Lambda writes these logs to CloudWatch Logs. Specific language statements generate log entries, depending on the language you use to author your Lambda function code.

Logging is subject to [CloudWatch Logs limits](#). Log data can be lost due to throttling or, in some cases, when the [execution context \(p. 104\)](#) is terminated.

- **Exceptions** – Your Lambda function needs to communicate the result of the function execution to AWS Lambda. Depending on the language you author your Lambda function code, there are different ways to end a request successfully or to notify AWS Lambda an error occurred during execution. If you invoke the function synchronously, then AWS Lambda forwards the result back to the client.
- **Concurrency** – When your function is invoked more quickly than a single instance of your function can process events, Lambda scales by running additional instances. Each instance of your function handles only one request at a time, so you don't need to worry about synchronizing threads or processes. You can, however, use asynchronous language features to process batches of events in parallel, and save data to the `/tmp` directory for use in future invocations on the same instance.

Your Lambda function code must be written in a stateless style, and have no affinity with the underlying compute infrastructure. Your code should expect local file system access, child processes, and similar artifacts to be limited to the lifetime of the request. Persistent state should be stored in Amazon S3, Amazon DynamoDB, or another cloud storage service. Requiring functions to be stateless enables AWS Lambda to launch as many copies of a function as needed to scale to the incoming rate of events and requests. These functions may not always run on the same compute instance from request to request, and a given instance of your Lambda function may be used more than once by AWS Lambda. For more information, see [Best Practices for Working with AWS Lambda Functions \(p. 127\)](#).

- [Building Lambda Functions with Node.js \(p. 241\)](#)
- [Building Lambda Functions with Python \(p. 251\)](#)
- [Building Lambda Functions with Ruby \(p. 329\)](#)
- [Building Lambda Functions with Java \(p. 263\)](#)
- [Building Lambda Functions with Go \(p. 290\)](#)
- [Building Lambda Functions with C# \(p. 302\)](#)
- [Building Lambda Functions with PowerShell \(p. 321\)](#)

Creating a Deployment Package

To create a Lambda function you first create a Lambda function deployment package, a .zip or .jar file consisting of your code and any dependencies. When creating the zip, include only the code and its dependencies, not the containing folder. You will then need to set the appropriate security permissions for the zip package.

- [AWS Lambda Deployment Package in Node.js \(p. 241\)](#)
- [AWS Lambda Deployment Package in Python \(p. 251\)](#)
- [AWS Lambda Deployment Package in Java \(p. 264\)](#)
- [AWS Lambda Deployment Package in Go \(p. 290\)](#)
- [AWS Lambda Deployment Package in C# \(p. 302\)](#)
- [AWS Lambda Deployment Package in PowerShell \(p. 321\)](#)

Permissions Policies on Lambda Deployment Packages

Zip packages uploaded with incorrect permissions may cause execution failure. AWS Lambda requires global read permissions on code files and any dependent libraries that comprise your deployment package. To ensure permissions are not restricted to your user account, you can check using the following samples:

- **Linux/Unix/OSX environments:** Use `zipinfo` as shown in the sample below:

```
$ zipinfo test.zip
Archive: test.zip
Zip file size: 473 bytes, number of entries: 2
-r----- 3.0 unix      0 bx stor 17-Aug-10 09:37 exlib.py
-r----- 3.0 unix     234 tx defN 17-Aug-10 09:37 index.py
2 files, 234 bytes uncompressed, 163 bytes compressed: 30.3%
```

The `-r-----` indicates that only the file owner has read permissions, which can cause Lambda function execution failures. The following indicates what you would see if there are requisite global read permissions:

```
$ zipinfo test.zip
Archive: test.zip
Zip file size: 473 bytes, number of entries: 2
-r--r--r-- 3.0 unx      0 bx stor 17-Aug-10 09:37 exlib.py
-r--r--r-- 3.0 unx     234 tx defN 17-Aug-10 09:37 index.py
2 files, 234 bytes uncompressed, 163 bytes compressed: 30.3%
```

To fix this recursively, run the following command:

```
$ chmod 644 $(find /tmp/package_contents -type f)
$ chmod 755 $(find /tmp/package_contents -type d)
```

- The first command changes all files in `/tmp/package_contents` to have read/write permissions to owners, read to group and global.
- The second command cascades the same permissions for directories.

Accessing AWS Resources from a Lambda Function

Lambda does not enforce any restrictions on your function logic – if you can code for it, you can run it within a Lambda function. As part of your function, you may need to call other APIs, or access other AWS services like databases.

Accessing AWS Services

To access other AWS services, you can use the AWS SDK. AWS Lambda automatically sets the credentials required by the SDK to those of the IAM role associated with your function – you do not need to take any additional steps. For example, here's sample code using the Python SDK for accessing an S3 object.

```
import boto3
import botocore

BUCKET_NAME = 'my-bucket' # replace with your bucket name
KEY = 'my_image_in_s3.jpg' # replace with your object key

s3 = boto3.resource('s3')

try:
    s3.Bucket(BUCKET_NAME).download_file(KEY, 'my_local_image.jpg')
except botocore.exceptions.ClientError as e:
    if e.response['Error']['Code'] == "404":
        print("The object does not exist.")
    else:
        raise
```

For convenience, AWS Lambda includes versions of the AWS SDK as part of the execution environment so you don't have to include it. See [AWS Lambda Runtimes \(p. 102\)](#) for the version of the included SDK. We recommend including your own copy of the AWS SDK for production applications so you can control your dependencies.

Accessing non AWS Services

You can include any SDK to access any service as part of your Lambda function. For example, you can include the [SDK for Twilio](#) to access information from your Twilio account. You can use [AWS Lambda](#)

[Environment Variables \(p. 40\)](#) for storing the credential information for the SDKs after encrypting the credentials.

Accessing Private Services or Resources

By default, your service or API must be accessible over the public internet for AWS Lambda to access it. However, you may have APIs or services that are not exposed this way. Typically, you create these resources inside Amazon Virtual Private Cloud (Amazon VPC) so that they cannot be accessed over the public Internet. These resources could be AWS service resources, such as Amazon Redshift data warehouses, Amazon ElastiCache clusters, or Amazon RDS instances. They could also be your own services running on your own EC2 instances. By default, resources within a VPC are not accessible from within a Lambda function.

AWS Lambda runs your function code securely within a VPC by default. However, to enable your Lambda function to access resources inside your private VPC, you must provide additional VPC-specific configuration information that includes VPC subnet IDs and security group IDs. AWS Lambda uses this information to set up elastic network interfaces (ENIs) that enable your function to connect securely to other resources within your private VPC.

Important

AWS Lambda does not support connecting to resources within Dedicated Tenancy VPCs. For more information, see [Dedicated VPCs](#).

To learn how to configure a Lambda function to access resources within a VPC, see [Configuring a Lambda Function to Access Resources in an Amazon VPC \(p. 68\)](#)

Configuring AWS Lambda Functions

You can use the AWS Lambda API or console to configure settings on your Lambda functions. [Basic function settings \(p. 35\)](#) include the description, role, and runtime that you specify when you create a function in the Lambda console. You can configure more settings after you create a function, or use the API to set things like the handler name, memory allocation, and security groups during creation.

To keep secrets out of your function code, store them in the function's configuration and read them from the execution environment during initialization. [Environment variables \(p. 40\)](#) are always encrypted at rest, and can be encrypted in transit as well. Use environment variables to make your function code portable by removing connection strings, passwords, and endpoints for external resources.

[Versions and aliases \(p. 47\)](#) are secondary resources that you can create to manage function deployment and invocation. Publish versions of your function to store its code and configuration as a separate resource that cannot be changed, and create an alias that points to a specific version. Then you can configure your clients to invoke a function alias, and update the alias when you want to point the client to a new version, instead of updating the client.

As you add libraries and other dependencies to your function, creating and uploading a deployment package can slow down development. Use [layers \(p. 64\)](#) to manage your function's dependencies independently and keep your deployment package small. You can also use layers to share your own libraries with other customers and use publicly available layers with your functions.

To use your Lambda function with AWS resources in an Amazon VPC, configure it with security groups and subnets to [create a VPC connection \(p. 68\)](#). Lambda uses [elastic network interfaces \(ENIs\)](#) to create the connection, so you need to ensure that your account has enough ENI capacity to handle the number of connections made as your function scales up under load.

Topics

- [AWS Lambda Function Configuration \(p. 35\)](#)
- [Managing Concurrency \(p. 37\)](#)
- [AWS Lambda Environment Variables \(p. 40\)](#)
- [AWS Lambda Function Versioning and Aliases \(p. 47\)](#)
- [AWS Lambda Layers \(p. 64\)](#)
- [Configuring a Lambda Function to Access Resources in an Amazon VPC \(p. 68\)](#)
- [Tagging Lambda Functions \(p. 76\)](#)

AWS Lambda Function Configuration

A Lambda function consists of code and any associated dependencies. In addition, a Lambda function also has configuration information associated with it. Initially, you specify the configuration information when you create a Lambda function.

To configure function settings

1. Open the [Lambda console](#).
2. Choose a function.
3. Configure any of the available options and then choose **Save**.

Function Settings

- **Code** – The code and dependencies of your function. For scripting languages, you can edit your function code in the embedded [editor \(p. 24\)](#). To add libraries, or for languages that the editor doesn't support, upload a [deployment package \(p. 32\)](#).
- **Runtime** – The [Lambda runtime \(p. 102\)](#) that executes your function.
- **Handler** – The method that the runtime executes when your function is invoked. The format for this value varies per language. See [Programming Model \(p. 31\)](#) for more information.
- **Environment variables** – Key-value pairs that Lambda sets in the execution environment. [Use environment variables \(p. 40\)](#) to extend your function's configuration outside of code.
- **Tags** – Key-value pairs that Lambda attaches to your function resource. [Use tags \(p. 76\)](#) to organize Lambda functions into groups for cost reporting and filtering in the Lambda console.

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

- **Execution role** – The [IAM role \(p. 9\)](#) that AWS Lambda assumes when it executes your function.
- **Description** – A description of the function.
- **Memory** – The amount of memory available to the function during execution. Choose an amount [between 128 MB and 3,008 MB \(p. 7\)](#) in 64 MB increments.

Lambda allocates CPU power linearly in proportion to the amount of memory configured. At 1,792 MB, a function has the equivalent of 1 full vCPU (one vCPU-second of credits per second).

- **Timeout** – The amount of time that Lambda allows a function to run before stopping it. The default is 3 seconds. The maximum allowed value is 900 seconds.
- **Virtual private cloud (VPC)** – If your function needs network access to resources that are not available over the internet, [configure it to connect to a VPC \(p. 68\)](#).
- **Dead letter queue (DLQ)** – If your function is invoked asynchronously, [choose a queue or topic \(p. 88\)](#) to receive failed invocations.
- **Enable active tracing** – Sample incoming requests and [trace sampled requests with AWS X-Ray \(p. 232\)](#).
- **Concurrency** – [Reserve concurrency for a function \(p. 37\)](#) to set the maximum number of simultaneous executions for a function, and reserves capacity for that concurrency level.

Reserved concurrency applies to the entire function, including all versions and aliases.

Function settings can only be changed on the unpublished version of a function. When you publish a version, code and most settings are locked to ensure a consistent experience for users of that version. Use [aliases \(p. 47\)](#) to propagate configuration changes in a controlled manner.

To configure functions with the Lambda API, use the following actions.

- [UpdateFunctionCode \(p. 456\)](#) – Update the function's code.
- [UpdateFunctionConfiguration \(p. 463\)](#) – Update version-specific settings.
- [TagResource \(p. 444\)](#) – Tag a function.
- [AddPermission \(p. 343\)](#) – Modify the [resource-based policy \(p. 10\)](#) of a function, version, or alias.
- [PutFunctionConcurrency \(p. 436\)](#) – Configure a function's reserved concurrency.
- [PublishVersion \(p. 430\)](#) – Create an immutable version with the current code and configuration.
- [CreateAlias \(p. 347\)](#) – Create aliases for function versions.

For example, to update a function's memory setting with the AWS CLI, use the `update-function-configuration` command.

```
$ aws lambda update-function-configuration --function-name my-function --memory-size 256
```

For function configuration best practices, see [Function Configuration \(p. 129\)](#).

Managing Concurrency

The unit of scale for AWS Lambda is a concurrent execution (see [Understanding Scaling Behavior \(p. 86\)](#) for more details). However, scaling indefinitely is not desirable in all scenarios. For example, you may want to control your concurrency for cost reasons, or to regulate how long it takes you to process a batch of events, or to simply match it with a downstream resource. To assist with this, Lambda provides a concurrent execution limit control at both the account level and the function level.

Account Level Concurrent Execution Limit

By default, AWS Lambda limits the total concurrent executions across all functions within a given region to 1000. You can view the account level setting by using the [GetAccountSettings \(p. 374\)](#) API and viewing the `AccountLimit` object. This limit can be raised as described below:

To request a limit increase for concurrent executions

1. Open the [AWS Support Center](#) page, sign in if necessary, and then choose **Create case**.
2. For **Regarding**, select **Service Limit Increase**.
3. For **Limit Type**, choose **Lambda**, fill in the necessary fields in the form, and then choose the button at the bottom of the page for your preferred method of contact.

Function Level Concurrent Execution Limit

By default, the concurrent execution limit is enforced against the sum of the concurrent executions of all functions. The shared concurrent execution pool is referred to as the unreserved concurrency allocation. If you haven't set up any function-level concurrency limit, then the unreserved concurrency limit is the same as the account level concurrency limit. Any increases to the account level limit have a corresponding increase in the unreserved concurrency limit. You can view the unreserved concurrency allocation for a function by using the [GetAccountSettings \(p. 374\)](#) API or using the AWS Lambda console. Functions that are being accounted against the shared concurrent execution pool will not show any concurrency value when queried using the [GetFunctionConfiguration \(p. 385\)](#) API.

You can optionally set the concurrent execution limit for a function. You may choose to do this for a few reasons:

- The default behavior means a surge of concurrent executions in one function prevents the function you have isolated with an execution limit from getting throttled. By setting a concurrent execution limit on a function, you are reserving the specified concurrent execution value for that function.
- Functions scale automatically based on incoming request rate, but not all resources in your architecture may be able to do so. For example, relational databases have limits on how many concurrent connections they can handle. You can set the concurrent execution limit for a function to align with the values of its downstream resources support.
- If your function connects to VPC based resources, you must make sure your subnets have adequate address capacity to support the ENI scaling requirements of your function. You can estimate the approximate ENI capacity with the following formula:

```
Concurrent executions * (Memory in GB / 3 GB)
```

Where:

- **Concurrent execution** – This is the projected concurrency of your workload. Use the information in [Understanding Scaling Behavior \(p. 86\)](#) to determine this value.
- **Memory in GB** – The amount of memory you configured for your Lambda function.

You can set the concurrent execution limit for a function to match the subnet size limits you have.

Note

If you need a function to stop processing any invocations, you can choose to set the concurrency to 0 and throttle all incoming executions.

By setting a concurrency limit on a function, Lambda guarantees that allocation will be applied specifically to that function, regardless of the amount of traffic processing remaining functions. If that limit is exceeded, the function will be throttled. How that function behaves when throttled will depend on the event source. For more information, see [Throttling Behavior \(p. 39\)](#).

Note

Concurrency limits can only be set at the function level, not for individual versions. All invocations to all versions and aliases of a given function will accrue towards the function limit.

Reserved vs. Unreserved Concurrency Limits

If you set the concurrent execution limit for a function, the value is deducted from the unreserved concurrency pool. For example, if your account's concurrent execution limit is 1000 and you have 10 functions, you can specify a limit on one function at 200 and another function at 100. The remaining 700 will be shared among the other 8 functions.

Note

AWS Lambda will keep the unreserved concurrency pool at a minimum of 100 concurrent executions, so that functions that do not have specific limits set can still process requests. So, in practice, if your total account limit is 1000, you are limited to allocating 900 to individual functions.

Setting Concurrency Limits Per Function (Console)

To set a concurrency limit for your Lambda function using the Lambda console, do the following:

1. Open the Lambda console [Functions page](#).
2. Choose a function.
3. Under the **Configuration** tab, choose **Concurrency**. In **Reserve concurrency**, set the value to the maximum of concurrent executions you want reserved for the function. Note that when you set this value, the **Unreserved account concurrency** value will automatically be updated to display the remaining number of concurrent executions available for all other functions in the account. Also note that if you want to block invocation of this function, set the value to 0. To remove a dedicated allotment value for this account, choose **Use unreserved account concurrency**.

Setting Concurrency Limits Per Function (CLI)

To set a concurrency limit for your Lambda function using the AWS CLI, do the following:

- Use the [PutFunctionConcurrency \(p. 436\)](#) operation and pass in the function name and concurrency limit you want allocated to this function:

```
$ aws lambda put-function-concurrency --function-name my-function --reserved-concurrent-executions 100
```

To remove a concurrency limit for your Lambda function using the AWS CLI, do the following:

- Use the [DeleteFunctionConcurrency \(p. 370\)](#) operation and pass in the function name:

```
$ aws lambda delete-function-concurrency --function-name my-function
```

To view a concurrency limit for your Lambda function using the AWS CLI, do the following:

- Use the [GetFunction \(p. 382\)](#) operation and pass in the function name:

```
$ aws lambda get-function --function-name my-function
```

Setting the per function concurrency can impact the concurrency pool available to other functions. We recommend restricting the permissions to the [PutFunctionConcurrency \(p. 436\)](#) API and [DeleteFunctionConcurrency \(p. 370\)](#) API to administrative users so that the number of users who can make these changes is limited.

Throttling Behavior

On reaching the concurrency limit associated with a function, any further invocation requests to that function are throttled, i.e. the invocation doesn't execute your function. Each throttled invocation increases the Amazon CloudWatch Throttles metric for the function. AWS Lambda handles throttled invocation requests differently, depending on their source:

- **Event sources that aren't stream-based:** Some of these event sources invoke a Lambda function synchronously, and others invoke it asynchronously. Handling is different for each:
 - **Synchronous invocation:** If the function is invoked synchronously and is throttled, Lambda returns a 429 error and the invoking service is responsible for retries. The `ThrottledReason` error code explains whether you ran into a function level throttle (if specified) or an account level throttle (see note below). Each service may have its own retry policy. For a list of event sources and their invocation type, see [Using AWS Lambda with Other Services \(p. 132\)](#).
 - **Asynchronous invocation:** If your Lambda function is invoked asynchronously and is throttled, AWS Lambda automatically retries the throttled event for up to six hours, with delays between retries.
- **Poll-based event sources that are also stream-based:** such as [Amazon Kinesis](#) or [Amazon DynamoDB](#), AWS Lambda polls your stream and invokes your Lambda function. When your Lambda function is throttled, Lambda attempts to process the throttled batch of records until the time the data expires. This time period can be up to seven days for Amazon Kinesis. The throttled request is treated as blocking per shard, and Lambda doesn't read any new records from the shard until the throttled batch of records either expires or succeeds. If there is more than one shard in the stream, Lambda continues invoking on the non-throttled shards until one gets through.
- **Poll-based event sources that are not stream-based:** such as [Amazon Simple Queue Service](#), AWS Lambda polls your queue and invokes your Lambda function. When your Lambda function is throttled, Lambda attempts to process the throttled batch of records until it is successfully invoked (in which case the message is automatically deleted from the queue) or until the `MessageRetentionPeriod` set for the queue expires.

Monitoring Your Concurrency Usage

To understand your concurrent execution usage, AWS Lambda provides the following metrics:

- **ConcurrentExecutions:** This shows you the concurrent executions at an account level, and for any function with a custom concurrency limit.

- **UnreservedConcurrentExecutions:** This shows you the total concurrent executions for functions assigned to the default unreserved concurrency pool.

To learn about these metrics and how to access them, see [Using Amazon CloudWatch \(p. 226\)](#).

AWS Lambda Environment Variables

Environment variables for Lambda functions enable you to dynamically pass settings 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, using either the AWS Lambda Console, the AWS Lambda CLI or the AWS Lambda SDK. AWS Lambda then makes these key value pairs available to your Lambda function code using standard APIs supported by the language, like `process.env` for Node.js functions.

You can use environment variables to help libraries know what directory to install files in, where to store outputs, store connection and logging settings, and more. By separating these settings from the application logic, you don't need to update your function code when you need to change the function behavior based on different settings.

Setting Up

Suppose you want a Lambda function to behave differently as it moves through lifecycle stages from development to deployment. For example, the dev, test, and production stages can contain databases that the function needs to connect to that require different connection information and use different table names. You can create environment variables to reference the database names, connection information or table names and set the value for the function based on the stage in which it's executing (for example, development, test, production) while your function code remains unchanged.

The following screenshots show how to modify your function's configuration using the AWS console. The first screenshot configures the settings for the function corresponding to a test stage. The second one configures settings for a production stage.

▼ Environment variables

You can define Environment Variables as key-value pairs that are accessible from your function code. These are useful to store configuration settings without the need to change function code. [Learn more.](#)

Database	Test_DB	Remove
DB_Connection	TEST	Remove
Key	Value	Remove

▼ Encryption configuration

Enable helpers for encryption in transit [Info](#)

KMS key to encrypt at rest [Info](#)

Select a KMS key to encrypt the environment variables at rest, or simply let Lambda manage the encryption.

(default) aws/lambda [▼](#) [Enter value](#)

You can define Environment Variables as key-value pairs that are accessible from your function code. These are useful to store configuration settings without the need to change function code. [Learn more](#).

Database	Prod_DB	Remove
DB_Connection	PROD	Remove
Key	Value	Remove

▼ **Encryption configuration**

Enable helpers for encryption in transit [Info](#)

KMS key to encrypt at rest [Info](#)
Select a KMS key to encrypt the environment variables at rest, or simply let Lambda manage the encryption.

(default) aws/lambda ▾ [Enter value](#)

Note the **Encryption configuration** section. You will learn more about using this in the [Create a Lambda Function Using Environment Variables To Store Sensitive Information \(p. 45\)](#) tutorial.

You can also use the AWS CLI to create Lambda functions that contain environment variables. For more details, see the [CreateFunction \(p. 355\)](#) and [UpdateFunctionConfiguration \(p. 463\)](#) APIs. Environment variables are also supported when creating and updating functions using AWS CloudFormation. Environment variables can also be used to configure settings specific to the language runtime or a library included in your function. For example, you can modify PATH to specify a directory where executables are stored. You can also set runtime-specific environment variables, such as PYTHONPATH for Python or NODE_PATH for Node.js.

The following example creates a new Lambda function that sets the LD_LIBRARY_PATH environment variable, which is used to specify a directory where shared libraries are dynamically loaded at runtime. In this example, the Lambda function code uses the shared library in the /usr/bin/test/lib64 directory. Note that the Runtime parameter uses nodejs6.10 but you can also specify nodejs8.10.

```
$ aws lambda create-function --function-name myTestFunction \
--zip-file fileb://package.zip \
--role role-arn \
--environment Variables="{LD_LIBRARY_PATH=/usr/bin/test/lib64}" \
--handler index.handler --runtime nodejs6.10
```

Rules for Naming Environment Variables

There is no limit to the number of environment variables you can create as long as the total size of the set does not exceed 4 KB.

Other requirements include:

- Must start with letters *[a-zA-Z]*.
- Can only contain alphanumeric characters and underscores (*[a-zA-Z0-9_]*).

In addition, there are a specific set of keys that AWS Lambda reserves. If you try to set values for any of these reserved keys, you will receive an error message indicating that the action is not allowed. For more information on these keys, see [Environment Variables Available to Lambda Functions \(p. 103\)](#).

Environment Variables and Function Versioning

Function versioning provides a way to manage your Lambda function code by enabling you to publish one or more versions of your Lambda function as it proceeds from development to test to production. For each version of a Lambda function that you publish, the environment variables (as well as other function-specific configurations such as `MemorySize` and `Timeout` limit) are saved as a snapshot of that version and those settings are immutable (cannot be changed).

As application and configuration requirements evolve, you can create new versions of your Lambda function and update the environment variables to meet those requirements prior to the newest version being published. The current version of your function is `$LATEST`.

In addition, you can create aliases, which are pointers to a particular version of your function. The advantage of aliases is that if you need to roll back to a previous function version, you point the alias to that version, which contains the environment variables required for that version. For more information, see [AWS Lambda Function Versioning and Aliases \(p. 47\)](#).

Environment Variable Encryption

When you create or update Lambda functions that use environment variables, AWS Lambda encrypts them using the [AWS Key Management Service](#). When your Lambda function is invoked, those values are decrypted and made available to the Lambda code.

The first time you create or update Lambda functions that use environment variables in a region, a default service key is created for you automatically within AWS KMS. This key is used to encrypt environment variables. However, should you wish to use encryption helpers and use KMS to encrypt environment variables after your Lambda function is created, then you must create your own AWS KMS key and choose it instead of the default key. The default key will give errors when chosen. Creating your own key gives you more flexibility, including the ability to create, rotate, disable, and define access controls, and to audit the encryption keys used to protect your data. For more information, see the [AWS Key Management Service Developer Guide](#).

If you use your own key, you will be billed per [AWS Key Management Service Pricing](#) guidelines. You will not be billed if you use the default service key provided by AWS Lambda.

If you're using the default KMS service key for Lambda, then no additional IAM permissions are required in your function execution role – your role will just work automatically without changes. If you're supplying your own (custom) KMS key, then you'll need to add `kms:Decrypt` to your execution role. In addition, the user that will be creating and updating the Lambda function must have permissions to use the KMS key. For more information on KMS keys, see the [Using Key Policies in AWS KMS](#).

Note

AWS Lambda authorizes your function to use the default KMS key through a user grant, which it adds when you assign the role to the function. If you delete the role and create a new role with the same name, you need to refresh the role's grant. Refresh the grant by re-assigning the role to the function.

Storing Sensitive Information

As mentioned in the previous section, when you deploy your Lambda function, all the environment variables you've specified are encrypted by default after, but not during, the deployment process. They

are then decrypted automatically by AWS Lambda when the function is invoked. If you need to store sensitive information in an environment variable, we strongly suggest you encrypt that information before deploying your Lambda function.

Fortunately, the Lambda console makes that easier for you by providing encryption helpers that leverage [AWS Key Management Service](#) to store that sensitive information as `Ciphertext`. The Lambda console also provides decryption helper code to decrypt that information for use in your Lambda function code. For more information, see [Create a Lambda Function Using Environment Variables To Store Sensitive Information \(p. 45\)](#).

Error scenarios

If your function configuration exceeds 4KB, or you use environment variable keys reserved by AWS Lambda, then your update or create operation will fail with a configuration error. During execution time, it's possible that the encryption/decryption of environment variables can fail. If AWS Lambda is unable to decrypt the environment variables due to an AWS KMS service exception, AWS KMS will return an exception message explaining what the error conditions are and what, if any, remedies you can apply to address the issue. These will be logged to your function log stream in Amazon CloudWatch logs. For example, if the KMS key you are using to access the environment variables is disabled, you will see the following error:

```
Lambda was unable to configure access to your environment variables because the KMS key used is disabled.  
Please check your KMS key settings.
```

Create a Lambda Function Using Environment Variables

This section will illustrate how you can modify a Lambda function's behavior through configuration changes that require no changes to the Lambda function code.

In this tutorial, you will do the following:

- Create a deployment package with sample code that returns the value of an environment variable that specifies the name of an Amazon S3 bucket.
- Invoke a Lambda function and verify that the Amazon S3 bucket name that is returned matches the value set by the environment variable.
- Update the Lambda function by changing the Amazon S3 bucket name specified by the environment variable.
- Invoke the Lambda function again and verify that the Amazon S3 bucket name that is returned matches the updated value.

Set Up the Lambda Environment

The code sample below reads the environment variable of a Lambda function that returns the name of an Amazon S3 bucket.

1. Open a text editor and copy the following code:

```
exports.handler = function(event, context, callback) {  
  var bucketName = process.env.S3_BUCKET;  
  callback(null, bucketName);  
};
```

2. Save the file as *index.js*.
3. Zip the *index.js*. file as *Test_Environment_Variables.zip*.

Create an Execution Role

Create an IAM role (execution role) that you can specify at the time you create your Lambda function.

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. Follow the steps in [IAM Roles](#) in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:
 - In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**.
 - In **Attach Policy**, choose the policy named **AWSLambdaBasicExecutionRole**.
3. Write down the Amazon Resource Name (ARN) of the IAM role. You need this value when you create your Lambda function in the next step.

Create the Lambda function and Test It

In this section, you create a Lambda function containing an environment variable that specifies an Amazon S3 bucket named `Test`. When invoked, the function simply returns the name of the Amazon S3 bucket. Then you update the configuration by changing the Amazon S3 bucket name to `Prod` and when invoked again, the function returns the updated name of the Amazon S3 bucket.

To create the Lambda function, open a command prompt and run the following Lambda AWS CLI `create-function` command. You need to provide the .zip file path and the execution role ARN.

```
$ aws lambda create-function --function-name ReturnBucketName \
--zip-file fileb://file-path/Test_Environment_Variables.zip \
--role role-arn \
--environment Variables='{S3_BUCKET=Test}' \
--handler index.handler --runtime nodejs8.10
```

Next, run the following Lambda CLI `invoke` command to invoke the function.

```
$ aws lambda invoke --function-name ReturnBucketName outputfile.txt
```

The Lambda function will return the name of the Amazon S3 bucket as "Test".

Next, run the following Lambda CLI `update-function-configuration` command to update the Amazon S3 environment variable by pointing it to the `Prod` bucket.

```
$ aws lambda update-function-configuration --function-name ReturnBucketName \
--environment Variables='{S3_BUCKET=Prod}'
```

Run the `aws lambda invoke` command again using the same parameters. This time, the Lambda function will return the Amazon S3 bucket name as `Prod`.

Create a Lambda Function Using Environment Variables To Store Sensitive Information

Along with specifying configuration settings for your Lambda function, you can also use environment variables to store sensitive information, such as a database password, using [AWS Key Management](#)

Service and the Lambda console's encryption helpers. For more information, see [Environment Variable Encryption \(p. 43\)](#). The following example shows you how to do this and also how to use KMS to decrypt that information.

This tutorial will demonstrate how you can use the Lambda console to encrypt an environment variable containing sensitive information. Note that if you are updating an existing function, you can skip ahead to the instruction step outlining how to Expand the **Environment Variables** section of [Step 2: Configure the Lambda Function \(p. 46\)](#).

Step 1: Create the Lambda Function

1. Sign in to the AWS Management Console and open the AWS Lambda console at <https://console.aws.amazon.com/lambda/>.
2. Choose **Create a Lambda function**.
3. In **Select blueprint**, choose the **Author from scratch** button.
4. In **Basic information**, do the following:
 - In **Name**, specify your Lambda function name.
 - In **Role**, choose **Choose an existing role**.
 - In **Existing role**, choose **lambda_basic_execution**.

Note

If the policy of the execution role does not have the `decrypt` permission, you will need add it.

- Choose **Create function**.

Step 2: Configure the Lambda Function

1. Under **Configuration**, specify the **Runtime** of your choice.
2. Under the **Lambda function code** section you can take advantage of the **Edit code inline** option to replace the Lambda function handler code with your custom code.
3. Note the **Triggers** tab. Under the **Triggers** page, you can optionally choose a service that automatically triggers your Lambda function by choosing the **Add trigger** button and then choosing the gray box with ellipses (...) to display a list of available services. For this example, do not configure a trigger and choose **Configuration**.
4. Note the **Monitoring** tab. This page will provide immediate CloudWatch metrics for your Lambda function invocations, as well as links to other helpful guides, including [Using AWS X-Ray \(p. 232\)](#).
5. Expand the **Environment variables** section.
6. Enter your key-value pair. Expand the **Encryption configuration** section. Note that Lambda provides a default service key under **KMS key to encrypt at rest** which encrypts your information after it has been uploaded. If the information you provided is sensitive, you can additionally check the **Enable helpers for encryption in transit** checkbox and supply a custom key. This masks the value you entered and results in a call to AWS KMS to encrypt the value and return it as **Ciphertext**. If you haven't created a KMS key for your account, you will be provided a link to the AWS IAM console to create one. The account must have `encrypt` and `decrypt` permissions for that key. Note that the **Encrypt** button toggles to **Decrypt** after you choose it. This affords you the option to update the information. Once you have done that, choose the **Encrypt** button.

The **Code** button provides sample decrypt code specific to the runtime of your Lambda function that you can use with your application.

Note

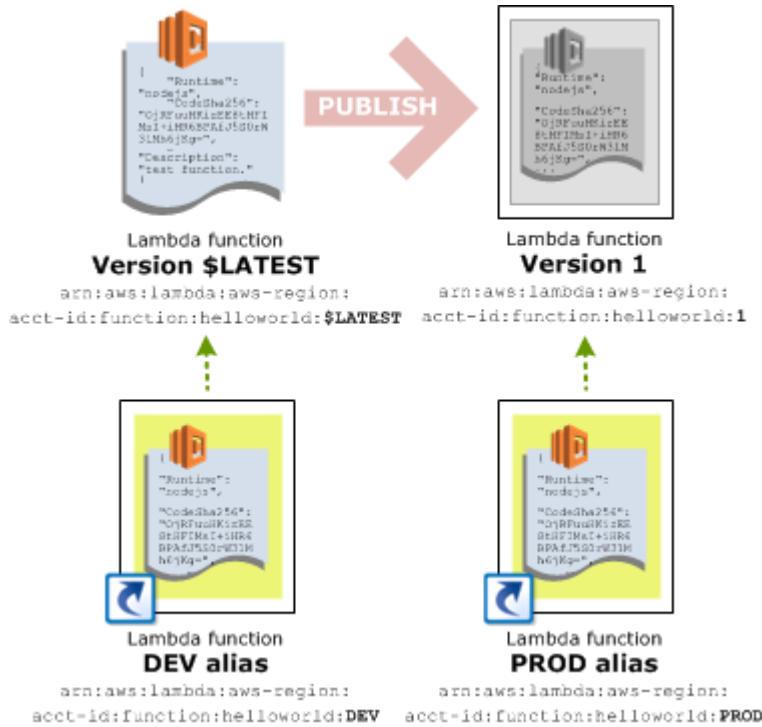
You cannot use the default Lambda service key for encrypting sensitive information on the client side. For more information, see [Environment Variable Encryption \(p. 43\)](#).

AWS Lambda Function Versioning and Aliases

By using versioning, you can manage your in-production function code in AWS Lambda better. When you use versioning in AWS Lambda, you can publish one or more versions of your Lambda function. As a result, you can work with different variations of your Lambda function in your development workflow, such as development, beta, and production.

Each Lambda function version has a unique Amazon Resource Name (ARN). After you publish a version, it can't be changed.

AWS Lambda also supports creating aliases for each of your Lambda function versions. Conceptually, an AWS Lambda alias is a pointer to a specific Lambda function version. It's also a resource similar to a Lambda function, and each alias has a unique ARN. Each alias maintains an ARN for the function version to which it points. An alias can only point to a function version, not to another alias. Unlike versions, aliases can be modified. You can update aliases to point to different versions of functions.



Aliases enable you to abstract the process of promoting new Lambda function versions into production from the mapping of the Lambda function version and its event source.

For example, suppose Amazon S3 is the event source that invokes your Lambda function when new objects are created in a bucket. When Amazon S3 is your event source, you store the event source mapping information in the bucket notification configuration. In that configuration, you can identify the Lambda function ARN that Amazon S3 can invoke. However, in this case each time you publish a new version of your Lambda function you need to update the notification configuration so that Amazon S3 invokes the correct version.

In contrast, instead of specifying the function ARN, suppose that you specify an alias ARN in the notification configuration (for example, PROD alias ARN). As you promote new versions of your Lambda function into production, you only need to update the PROD alias to point to the latest stable version. You don't need to update the notification configuration in Amazon S3.

The same applies when you need to roll back to a previous version of your Lambda function. In this scenario, you just update the PROD alias to point to a different function version. There is no need to update event source mappings.

We recommend that you use versioning and aliases to deploy your Lambda functions when building applications with multiple dependencies and developers involved.

Topics

- [Introduction to AWS Lambda Versioning \(p. 48\)](#)
- [Introduction to AWS Lambda Aliases \(p. 51\)](#)
- [Versioning, Aliases, and Resource Policies \(p. 58\)](#)
- [Managing Versioning Using the AWS Management Console, the AWS CLI, or Lambda API Operations \(p. 59\)](#)
- [Traffic Shifting Using Aliases \(p. 62\)](#)

Introduction to AWS Lambda Versioning

Following, you can find how to create a Lambda function and publish a version from it. You can also find how to update function code and configuration information when you have one or more published versions. In addition, you can find information on how to delete function versions, either specific versions or an entire Lambda function with all of its versions and associated aliases.

Creating a Lambda Function (the \$LATEST Version)

When you create a Lambda function, there is only one version—the \$LATEST version.



You can refer to this function using its Amazon Resource Name (ARN). There are two ARNs associated with this initial version:

- **Qualified ARN** – The function ARN with the version suffix.

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:$LATEST
```

- **Unqualified ARN** – The function ARN without the version suffix.

You can use this unqualified ARN in all relevant operations. However, you cannot use it to create an alias. For more information, see [Introduction to AWS Lambda Aliases \(p. 51\)](#).

The unqualified ARN has its own resource policies.

```
arn:aws:lambda:aws-region:acct-id:function:helloworld
```

Note

Unless you choose to publish versions, the \$LATEST function version is the only Lambda function version that you have. You can use either the qualified or unqualified ARN in your event source mapping to invoke the \$LATEST version.

The following is an example response for a CreateFunction API call.

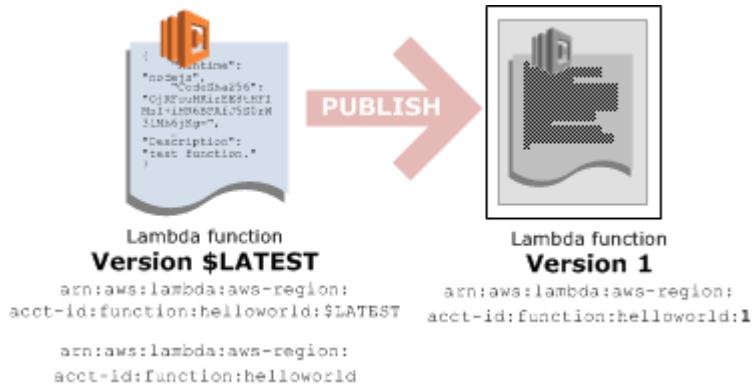
```
{
    "CodeSize": 287,
    "Description": "test function.",
    "FunctionArn": "arn:aws:lambda:aws-region:acct-id:function:helloworld",
    "FunctionName": "helloworld",
    "Handler": "helloworld.handler",
    "LastModified": "2015-07-16T00:34:31.322+0000",
    "MemorySize": 128,
    "Role": "arn:aws:iam::acct-id:role/lambda_basic_execution",
    "Runtime": "nodejs6.10",
    "Timeout": 3,
    "CodeSHA256": "OjRFuuHKizEE8tHFIMsI+iHR6BPAfJ5S0rW31Mh6jKg=",
    "Version": "$LATEST"
}
```

For more information, see [CreateFunction \(p. 355\)](#).

In this response, AWS Lambda returns the unqualified ARN of the newly created function and also its version, \$LATEST. The response also shows that the Version is \$LATEST. The CodeSha256 is the checksum of the deployment package that you uploaded.

Publishing an AWS Lambda Function Version

When you publish a version, AWS Lambda makes a snapshot copy of the Lambda function code (and configuration) in the \$LATEST version. A published version is immutable. That is, you can't change the code or configuration information. The new version has a unique ARN that includes a version number suffix as shown following.



You can publish a version by using any of the following methods:

- **Publish a version explicitly** – You can use the PublishVersion API operation to explicitly publish a version. For more information, see [PublishVersion \(p. 430\)](#). This operation creates a new version using the code and configuration in the \$LATEST version.
- **Publish a version at the time you create or update a Lambda function** – You can also use the CreateFunction or UpdateFunctionCode requests to publish a version by adding the optional publish parameter in the request:
 - Specify the publish parameter in your CreateFunction request to create a new Lambda function (the \$LATEST version). You can then immediately publish the new function by creating

a snapshot and assigning it to be version 1. For more information about `CreateFunction`, see [CreateFunction \(p. 355\)](#).

- Specify the `publish` parameter in your `UpdateFunctionCode` request to update the code in the `$LATEST` version. You can then publish a version from the `$LATEST`. For more information about `UpdateFunctionCode`, see [UpdateFunctionCode \(p. 456\)](#).

If you specify the `publish` parameter at the time you create a Lambda function, the function configuration information that AWS Lambda returns in response shows the version number of the newly published version. In the following example, the version is 1.

```
{  
    "CodeSize": 287,  
    "Description": "test function."  
    "FunctionArn": "arn:aws:lambda:aws-region:acct-id:function:helloworld",  
    "FunctionName": "helloworld",  
    "Handler": "helloworld.handler",  
    "LastModified": "2015-07-16T00:34:31.322+0000",  
    "MemorySize": 128,  
    "Role": "arn:aws:iam::acct-id:role/lambda_basic_execution",  
    "Runtime": "nodejs6.10",  
    "Timeout": 3,  
    "CodeSHA256": "OjRFuuHKizEE8tHFIMsI+iHR6BPAfJ5S0rW31Mh6jKg=",  
    "Version": "1"  
}
```

Note

Lambda only publishes a new version if the code hasn't yet been published or if the code has changed when compared against the `$LATEST` version. If there is no change, the `$LATEST` published version is returned.

We recommend that you publish a version at the same time that you create your Lambda function or update your Lambda function code. This recommendation especially applies when multiple developers contribute to the same Lambda function development. You can use the `publish` parameter in your request to do this.

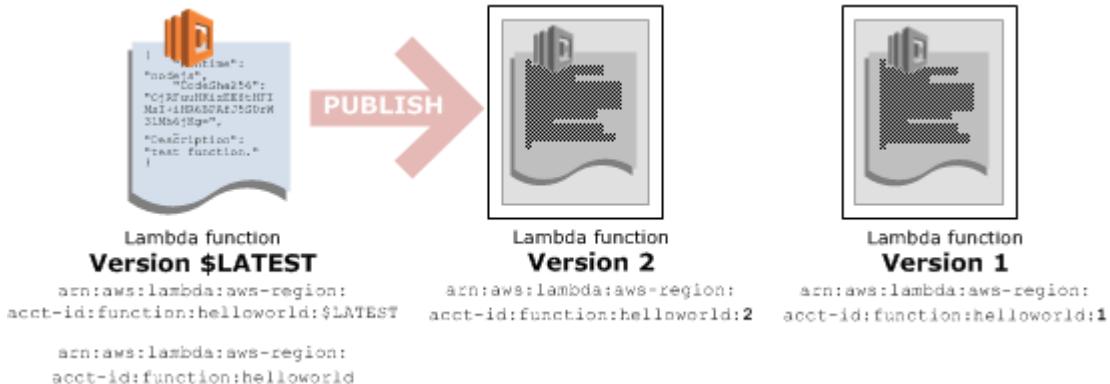
When you have multiple developers working on a project, you can have a scenario where developer A creates a Lambda function (`$LATEST` version). Before developer A publishes this version, developer B might update the code (the deployment package) associated with the `$LATEST` version. In this case, you lose the original code that developer A uploaded. When both developers add the `publish` parameter, it prevents the race condition described.

Each version of a Lambda function is a unique resource with a Amazon Resource Name (ARN). The following example shows the ARN of version number 1 of the `helloworld` Lambda function.

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:1
```

You can publish multiple versions of a Lambda function. Each time you publish a version, AWS Lambda copies `$LATEST` version (code and configuration information) to create a new version. When you publish additional versions, AWS Lambda assigns a monotonically increasing sequence number for versioning, even if the function was deleted and recreated. Version numbers are never reused, even for a function that has been deleted and recreated. This approach means that the consumer of a function version can depend on the executable of that version to never change (except if it's deleted).

If you want to reuse a qualifier, use aliases with your versions. Aliases can be deleted and re-created with the same name.



Updating Lambda Function Code and Configuration

AWS Lambda maintains your latest function code in the `$LATEST` version. When you update your function code, AWS Lambda replaces the code in the `$LATEST` version of the Lambda function. For more information, see [UpdateFunctionCode \(p. 456\)](#).

You have the following options of publishing a new version as you update your Lambda function code:

- Publish a version in the same update code request** – Use the `UpdateFunctionCode` API operation (recommended).
- First update the code, and then explicitly publish a version** – Use the `PublishVersion` API operation.

You can update code and configuration information (such as description, memory size, and execution timeout) for the `$LATEST` version of the Lambda function.

Deleting a Lambda Function and a Specific Version

With versioning, you have the following choices:

- Delete a specific version** – You can delete a Lambda function version by specifying the version you want to delete in your `DeleteFunction` request. If there are aliases that depend on this version, the request fails. AWS Lambda deletes the version only if there are no aliases dependent on this version. For more information about aliases, see [Introduction to AWS Lambda Aliases \(p. 51\)](#).
- Delete the entire Lambda function (all of its versions and aliases)** – To delete the Lambda function and all of its versions, don't specify any version in your `DeleteFunction` request. Doing this deletes the entire function including all of its versions and aliases.

Introduction to AWS Lambda Aliases

You can create one or more aliases for your Lambda function. An AWS Lambda alias is like a pointer to a specific Lambda function version. For more information about versioning, see [Introduction to AWS Lambda Versioning \(p. 48\)](#).

By using aliases, you can access the Lambda function an alias is pointing to (for example, to invoke the function) without the caller having to know the specific version the alias is pointing to.

AWS Lambda aliases enable the following use cases:

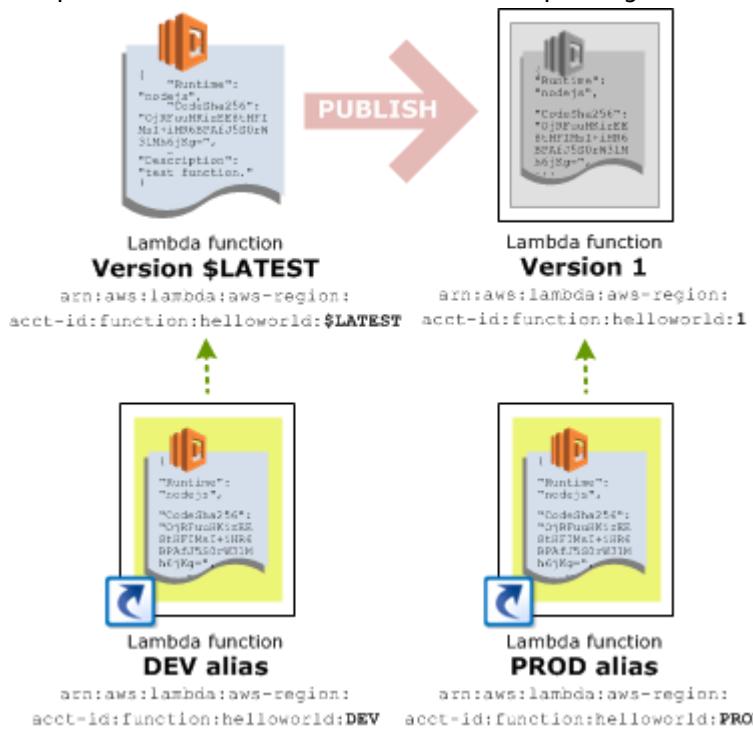
- Easier support for promotion of new versions of Lambda functions and rollback when needed** – After initially creating a Lambda function (the `$LATEST` version), you can publish a version 1 of it.

By creating an alias named PROD that points to version 1, you can now use the PROD alias to invoke version 1 of the Lambda function.

Now, you can update the code (the `$LATEST` version) with all of your improvements, and then publish another stable and improved version (version 2). You can promote version 2 to production by remapping the PROD alias so that it points to version 2. If you find something wrong, you can easily roll back the production version to version 1 by remapping the PROD alias so that it points to version 1.

- **Simplify management of event source mappings** – Instead of using Amazon Resource Names (ARNs) for Lambda function in event source mappings, you can use an alias ARN. This approach means that you don't need to update your event source mappings when you promote a new version or roll back to a previous version.

Both the Lambda function and alias are AWS Lambda resources, and like all other AWS resources they both have unique ARNs. The following example shows a Lambda function (the `$LATEST` version), with one published version. Each version has an alias pointing to it.



You can access the function using either the function ARN or the alias ARN.

- Because the function version for an unqualified function always maps to `$LATEST`, you can access it using the qualified or unqualified function ARN. The following shows a qualified function ARN with the `$LATEST` version suffix.

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:$LATEST
```

- When using any of the alias ARNs, you are using a qualified ARN. Each alias ARN has an alias name suffix.

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:PROD
arn:aws:lambda:aws-region:acct-id:function:helloworld:BETA
arn:aws:lambda:aws-region:acct-id:function:helloworld:DEV
```

AWS Lambda provides the following API operations for you to create and manage aliases:

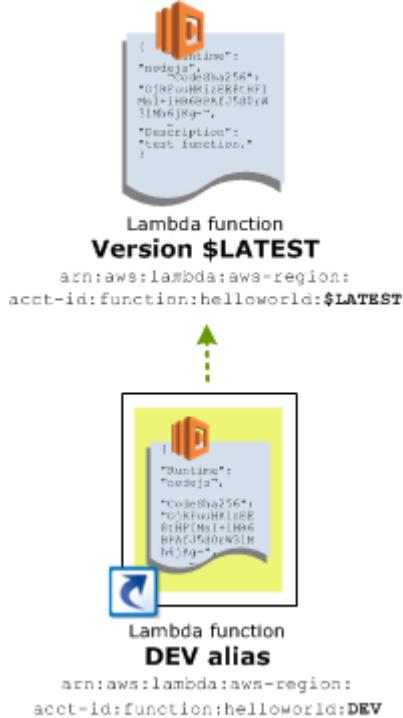
- [CreateAlias \(p. 347\)](#)
- [UpdateAlias \(p. 448\)](#)
- [GetAlias \(p. 376\)](#)
- [ListAliases \(p. 407\)](#)
- [DeleteAlias \(p. 363\)](#)

Example: Using Aliases to Manage Lambda Function Versions

The following is an example scenario of how to use versioning and aliases to promote new versions of Lambda functions into production.

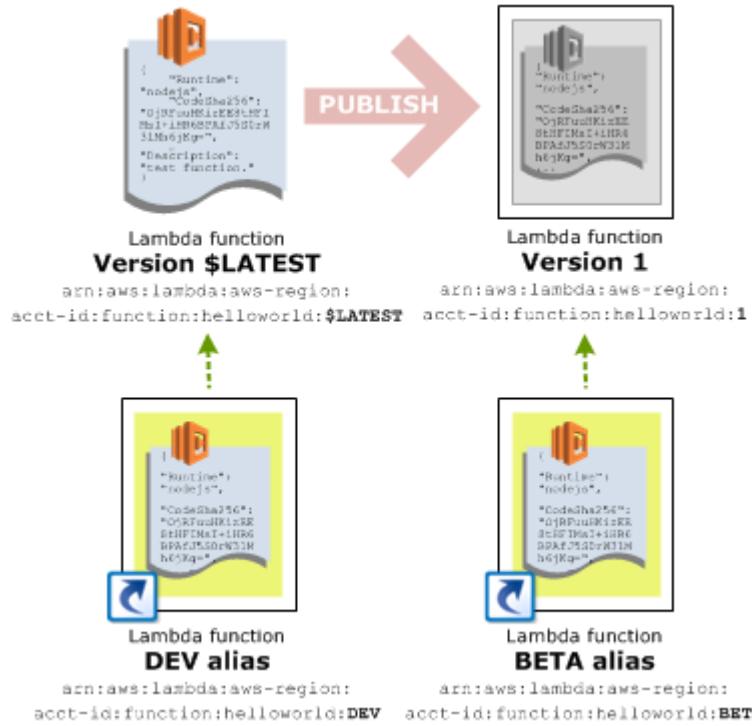
Initially, you create a Lambda function.

The function you create is the `$LATEST` version. You also create an alias (DEV, for development) that points to the newly created function. Developers can use this alias to test the function with the event sources in a development environment.



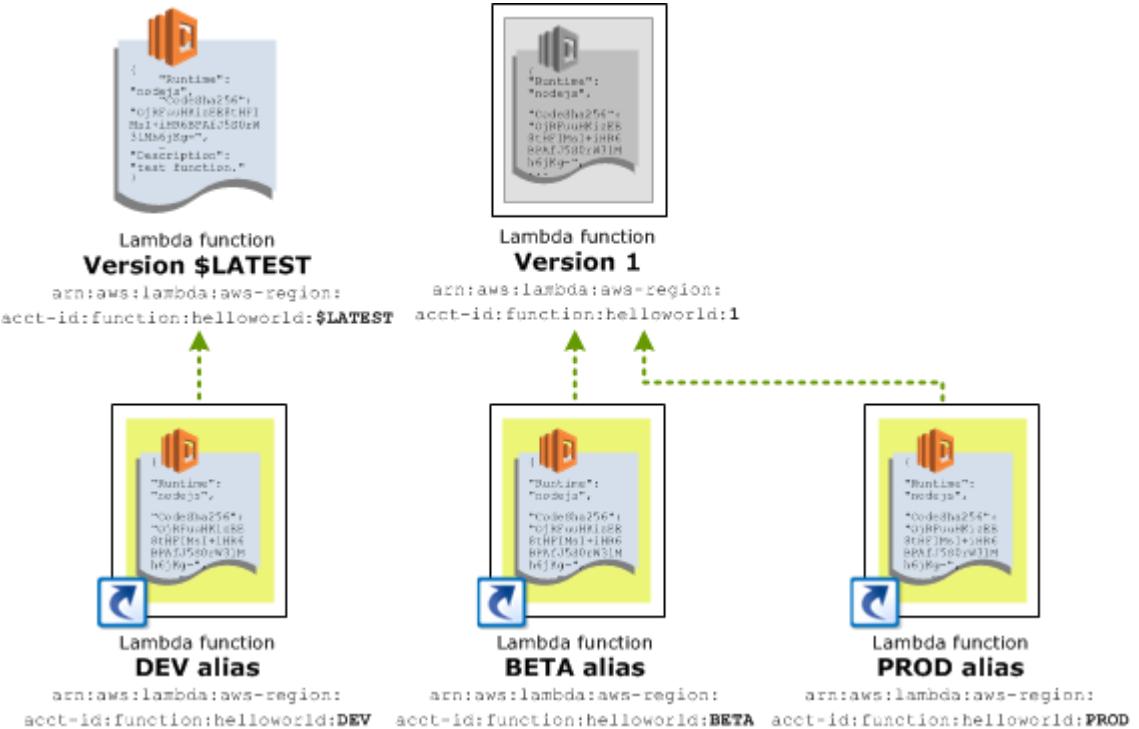
You then test the function version using event sources in a beta environment in a stable way, while continuing to develop newer versions.

You publish a version from the `$LATEST` and have another alias (BETA) point to it. This approach allows you to associate your beta event sources to this specific alias. In the event source mappings, use the BETA alias to associate your Lambda function with the event source.



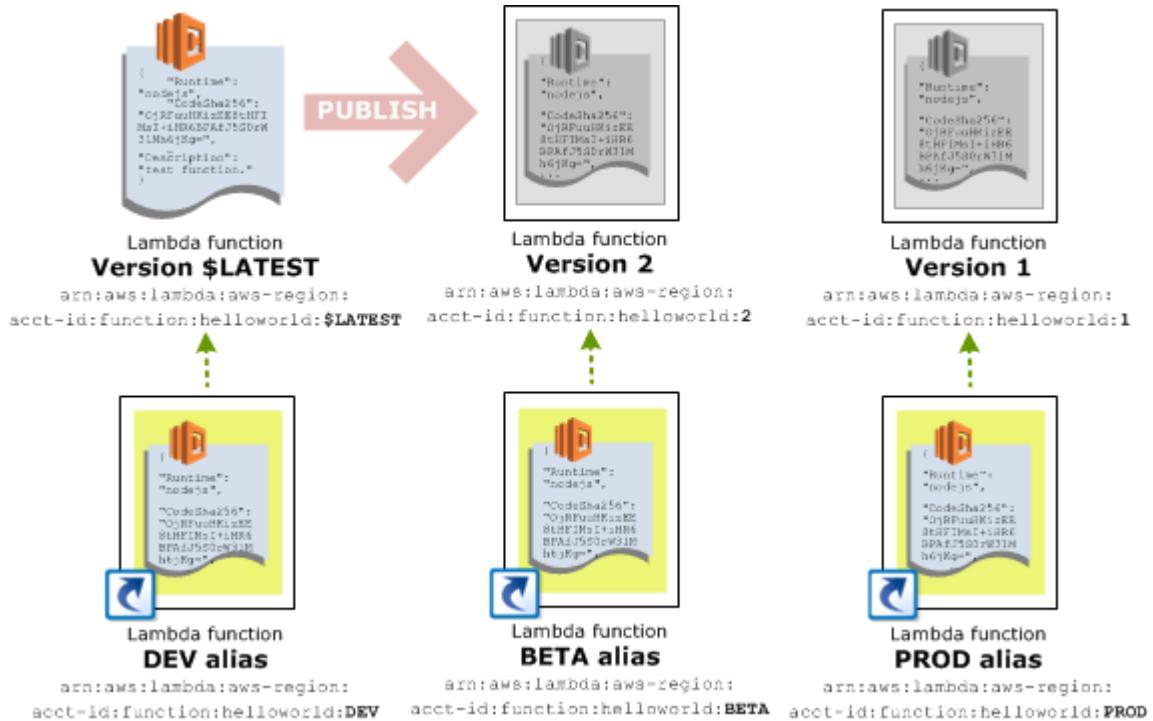
You then promote the Lambda function version in production to work with event sources in production environment.

After testing the BETA version of the function, you can define the production version by creating an alias that maps to version 1. In this approach, you point your production event sources to this specific version. You do this by creating a PROD alias and using the PROD alias ARN in all of your production event source mappings.



You continue development, publish more versions, and test.

As you develop your code, you can update the `$LATEST` version by uploading updated code and then publish to beta testing by having the `BETA` alias point to it. This simple remapping of the beta alias lets you put version 2 of your Lambda function into beta without changing any of your event sources. This approach is how aliases enable you to control which versions of your function are used with specific event sources in your development environment.



If you want to try creating this setup using the AWS Command Line Interface, see [Tutorial: Using AWS Lambda Aliases \(p. 55\)](#).

Tutorial: Using AWS Lambda Aliases

This AWS CLI-based tutorial creates Lambda function versions and aliases that point to it as described in the [Example: Using Aliases to Manage Lambda Function Versions \(p. 53\)](#).

This example uses the `us-west-2` (US West Oregon) region to create the Lambda function and aliases.

1. Create a deployment package that you can upload to create your Lambda function:

- a. Open a text editor, and then copy the following code.

```
console.log('Loading function');

exports.handler = function(event, context, callback) {
    console.log('value1 =', event.key1);
    console.log('value2 =', event.key2);
    console.log('value3 =', event.key3);
    callback(null, "message");
};
```

- b. Save the file as `helloworld.js`.

- c. Zip the `helloworld.js` file as `helloworld.zip`.
2. Create an AWS Identity and Access Management (IAM) role (execution role) that you can specify at the time you create your Lambda function:
 - a. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
 - b. Follow the steps in [IAM Roles](#) in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:
 - For **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**.
 - For **Attach Policy**, choose the policy named **AWSLambdaBasicExecutionRole**.
 - c. Write down the Amazon Resource Name (ARN) of the IAM role. You need this value when you create your Lambda function in the next step.
3. Create a Lambda function (`helloworld`).

```
aws lambda create-function --function-name helloworld --runtime nodejs6.10 \
--zip-file fileb://helloworld.zip --handler helloworld.handler \
--role arn:aws:iam::account-id:role/lambda_basic_execution \
```

The response returns the configuration information showing `$LATEST` as the function version as shown in the following example response.

```
{
  "CodeSha256": "OjRFuuHKizEE8tHFIMsI+iHR6BPAfJ5S0rW31Mh6jKg=",
  "FunctionName": "helloworld",
  "CodeSize": 287,
  "MemorySize": 128,
  "FunctionArn": "arn:aws:lambda:us-west-2:account-id:function:helloworld",
  "Version": "$LATEST",
  "Role": "arn:aws:iam::account-id:role/lambda_basic_execution",
  "Timeout": 3,
  "LastModified": "2015-09-30T18:39:53.873+0000",
  "Handler": "helloworld.handler",
  "Runtime": "nodejs6.10",
  "Description": ""
}
```

4. Create an alias (`DEV`) that points to the `$LATEST` version of the `helloworld` Lambda function.

```
aws lambda create-alias --function-name helloworld --name DEV \
--description "sample alias" --function-version "\$LATEST"
```

The response returns the alias information, including the function version it points to and the alias ARN. The ARN is the same as the function ARN with an alias name suffix. The following is an example response.

```
{
  "AliasArn": "arn:aws:lambda:us-west-2:account-id:function:helloworld:DEV",
  "FunctionVersion": "$LATEST",
  "Name": "DEV",
  "Description": "sample alias"
}
```

5. Publish a version of the `helloworld` Lambda function.

```
aws lambda publish-version --function-name helloworld
```

The response returns configuration information of the function version, including the version number, and the function ARN with the version suffix. The following is an example response.

```
{  
    "CodeSha256": "OjRFuuHKizeE8tHFIMsI+iHR6BPAfJ5S0rW31Mh6jKg=",  
    "FunctionName": "helloworld",  
    "CodeSize": 287,  
    "MemorySize": 128,  
    "FunctionArn": "arn:aws:lambda:us-west-2:account-id:function:helloworld:1",  
    "Version": "1",  
    "Role": "arn:aws:iam::account-id:role/lambda_basic_execution",  
    "Timeout": 3,  
    "LastModified": "2015-10-03T00:48:00.435+0000",  
    "Handler": "helloworld.handler",  
    "Runtime": "nodejs6.10",  
    "Description": ""  
}
```

6. Create an alias named **BETA** for the **helloworld** Lambda function version 1.

```
aws lambda create-alias --function-name helloworld \  
--description "sample alias" --function-version 1 --name BETA
```

Now you have two aliases for the **helloworld** function. The **DEV** alias points to the **\$LATEST** function version, and the **BETA** alias points to version 1 of the Lambda function.

7. Suppose that you want to put the version 1 of the **helloworld** function in production. Create another alias (**PROD**) that points to version 1.

```
aws lambda create-alias --function-name helloworld \  
--description "sample alias" --function-version 1 --name PROD
```

At this time, you have both the **BETA** and **PROD** aliases pointing to version 1 of the Lambda function.

8. You can now publish a newer version (for example, version 2), but first you need to update your code and upload a modified deployment package. If the **\$LATEST** version is not changed, you cannot publish more than one version of it. Assuming you updated the deployment package, uploaded it, and published version 2, you can now change the **BETA** alias to point to version 2 of the Lambda function.

```
aws lambda update-alias --function-name helloworld \  
--function-version 2 --name BETA
```

Now you have three aliases pointing to a different version of the Lambda function (**DEV** alias points to the **\$LATEST** version, **BETA** alias points to version 2, and the **PROD** alias points to version 1 of the Lambda function).

For information about using the AWS Lambda console to manage versioning, see [Managing Versioning Using the AWS Management Console, the AWS CLI, or Lambda API Operations \(p. 59\)](#).

Granting Permissions in a Push Model

In a push model (see [AWS Lambda Event Source Mapping \(p. 82\)](#)), event sources such as Amazon S3 invoke your Lambda function. These event sources maintain a mapping that identifies the function version or alias that they invoke when events occur. Note the following:

- We recommend that you specify an existing Lambda function alias in the mapping configuration (see [Introduction to AWS Lambda Aliases \(p. 51\)](#)). For example, if the event source is Amazon S3, you

specify the alias ARN in the bucket notification configuration so that Amazon S3 can invoke the alias when it detects specific events.

- In the push model, you grant event sources permissions using a resource policy that you attach to your Lambda function. In versioning, the permissions you add are specific to the qualifier that you specify in the `AddPermission` request (see [Versioning, Aliases, and Resource Policies \(p. 58\)](#)).

For example, the following AWS CLI command grants Amazon S3 permissions to invoke the PROD alias of the helloworld Lambda function (note that the `--qualifier` parameter specifies the alias name).

```
aws lambda add-permission --function-name helloworld \
--qualifier PROD --statement-id 1 --principal s3.amazonaws.com --action
lambda:InvokeFunction \
--source-arn arn:aws:s3:::examplebucket --source-account 111111111111
```

In this case, Amazon S3 is now able to invoke the PROD alias and AWS Lambda can then execute the helloworld Lambda function version that the PROD alias points to. For this to work, you must use the PROD alias ARN in the S3 bucket's notification configuration.

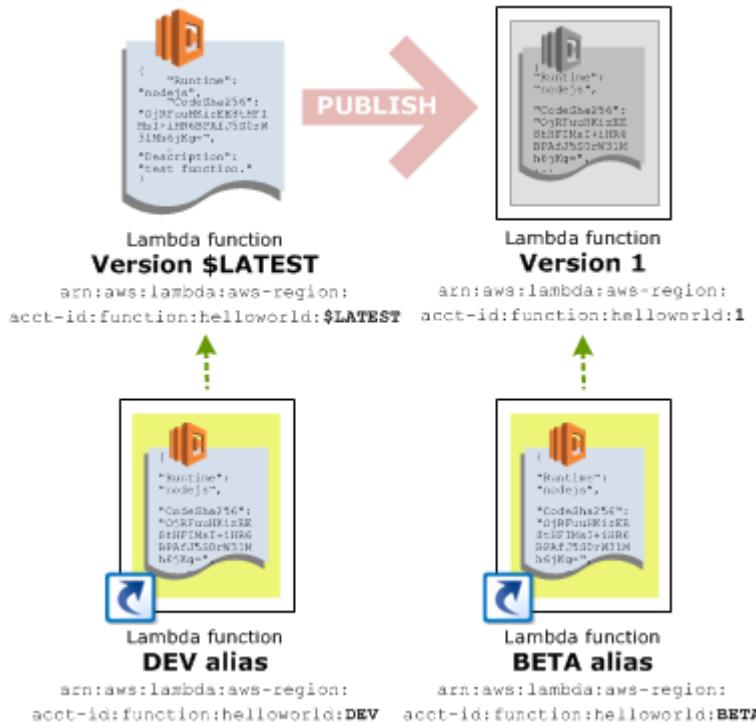
For information about how to handle Amazon S3 events, see [Tutorial: Using AWS Lambda with Amazon S3 \(p. 196\)](#).

Note

If you use the AWS Lambda console to add an event source for your Lambda function, the console adds the necessary permissions for you.

Versioning, Aliases, and Resource Policies

With versioning and aliases you can access a Lambda function using various ARNs. For example, consider the following scenario.



You can invoke for example the `helloworld` function version 1 using any of the following two ARNs:

- Using the qualified function ARN as shown following.

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:1
```

- Using the BETA alias ARN as shown following.

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:BETA
```

In a *push* model, event sources (such as Amazon S3 and custom applications) can invoke any of the Lambda function versions as long you grant the necessary permissions to these event sources by using an access policy associated with the Lambda function. For more information about the push model, see [AWS Lambda Event Source Mapping \(p. 82\)](#).

Assuming that you grant permission, the next question is, "can an event source invoke a function version using any of the associated ARNs?" The answer is, it depends on how you identified function in your add permissions request (see [AddPermission \(p. 343\)](#)). The key to understanding this is that the permission you grant apply only to the ARN used in the add permission request:

- If you use a qualified function name (such as `helloworld:1`), the permission is valid for invoking the `helloworld` function version 1 *only* using its qualified ARN (using any other ARNs results in a permission error).
- If you use an alias name (such as `helloworld:BETA`), the permission is valid only for invoking the `helloworld` function using the BETA alias ARN (using any other ARNs results in a permission error, including the function version ARN to which the alias points).
- If you use an unqualified function name (such as `helloworld`), the permission is valid only for invoking the `helloworld` function using the unqualified function ARN (using any other ARNs will result in a permission error).

Note

Note that even though the access policy is only on the unqualified ARN, the code and configuration of the invoked Lambda function is still from function version `$LATEST`. The unqualified function ARN maps to the `$LATEST` version but the permissions you add are ARN-specific.

- If you use a qualified function name using the `$LATEST` version (`helloworld:$LATEST`), the permission is valid for invoking the `helloworld` function version `$LATEST` *only* using its qualified ARN (using unqualified ARN results in a permission error).

Managing Versioning Using the AWS Management Console, the AWS CLI, or Lambda API Operations

You can manage Lambda function versioning programmatically using AWS SDKs (or make the AWS Lambda API calls directly, if you need to), using AWS Command Line Interface (AWS CLI), or the AWS Lambda console.

AWS Lambda provides the following APIs to manage versioning and aliases:

[PublishVersion \(p. 430\)](#)

[ListVersionsByFunction \(p. 423\)](#)

[CreateAlias \(p. 347\)](#)

[UpdateAlias \(p. 448\)](#)

[DeleteAlias \(p. 363\)](#)

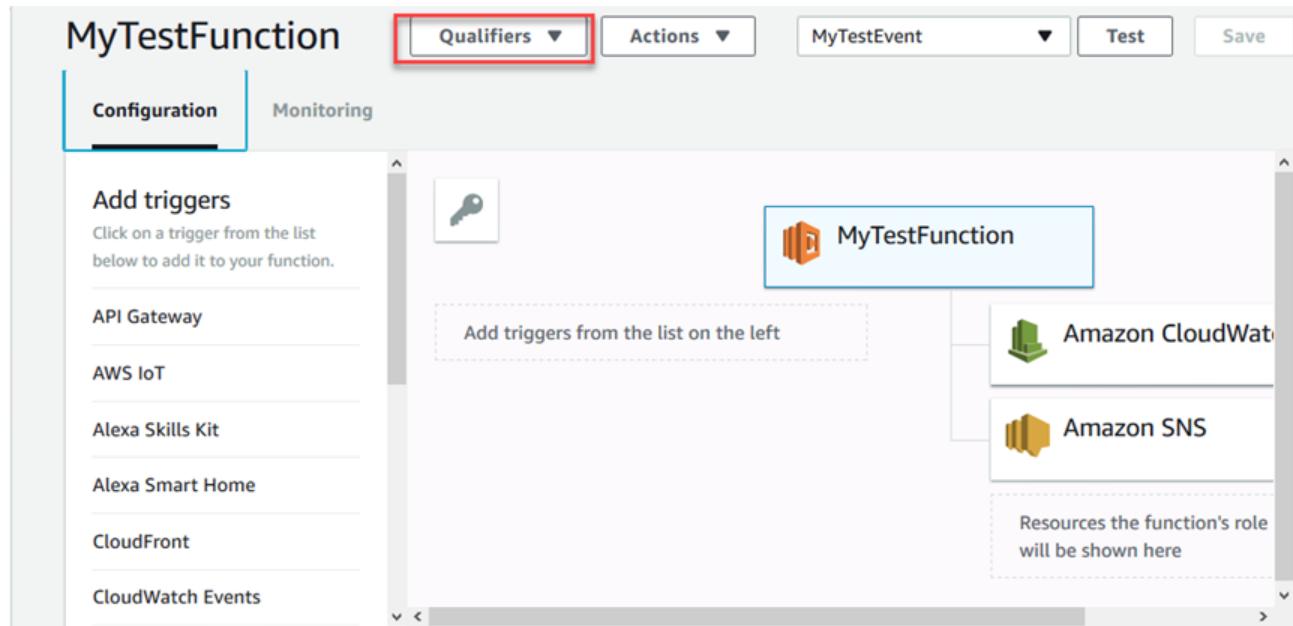
[GetAlias \(p. 376\)](#)

[ListAliases \(p. 407\)](#)

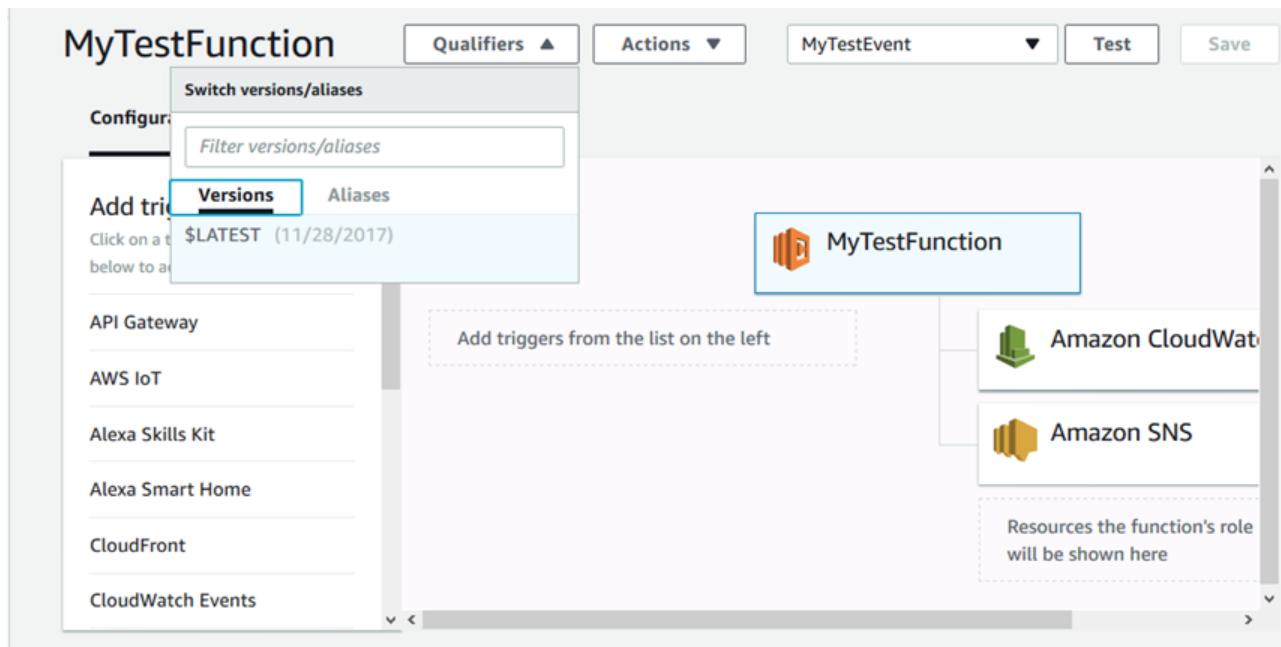
In addition to these APIs, existing relevant APIs also support versioning related operations.

For an example of how you can use the AWS CLI, see [Tutorial: Using AWS Lambda Aliases \(p. 55\)](#).

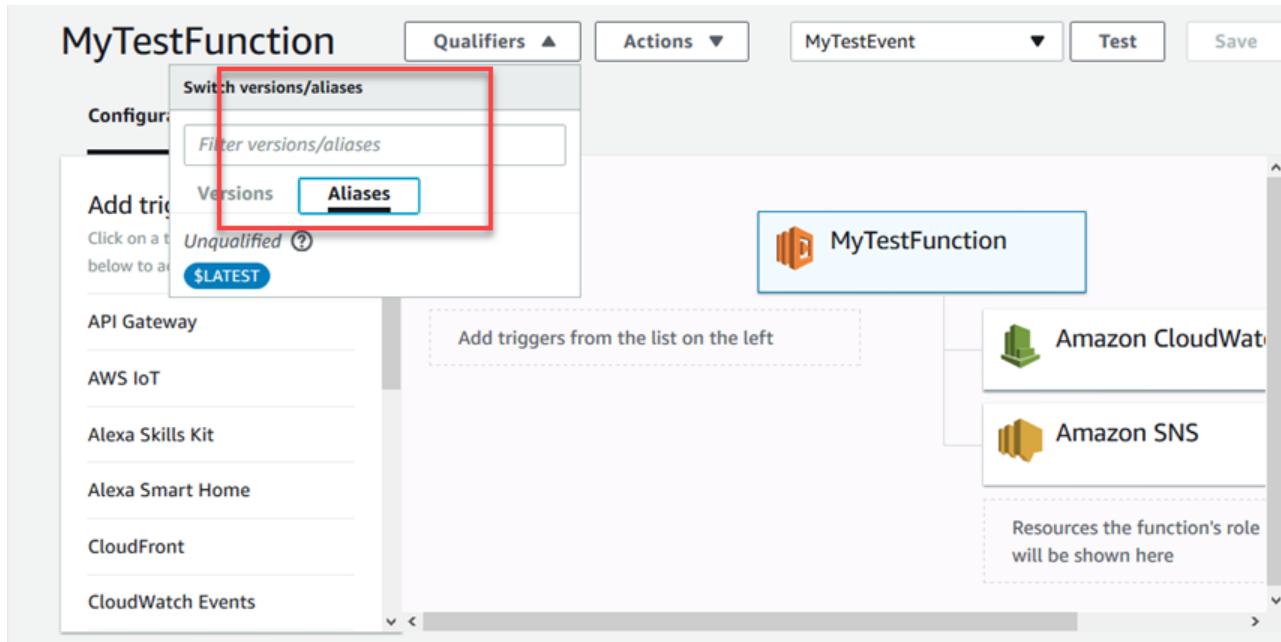
This section explains how you can use the AWS Lambda console to manage versioning. In the AWS Lambda console, choose a function and then choose **Qualifiers**.



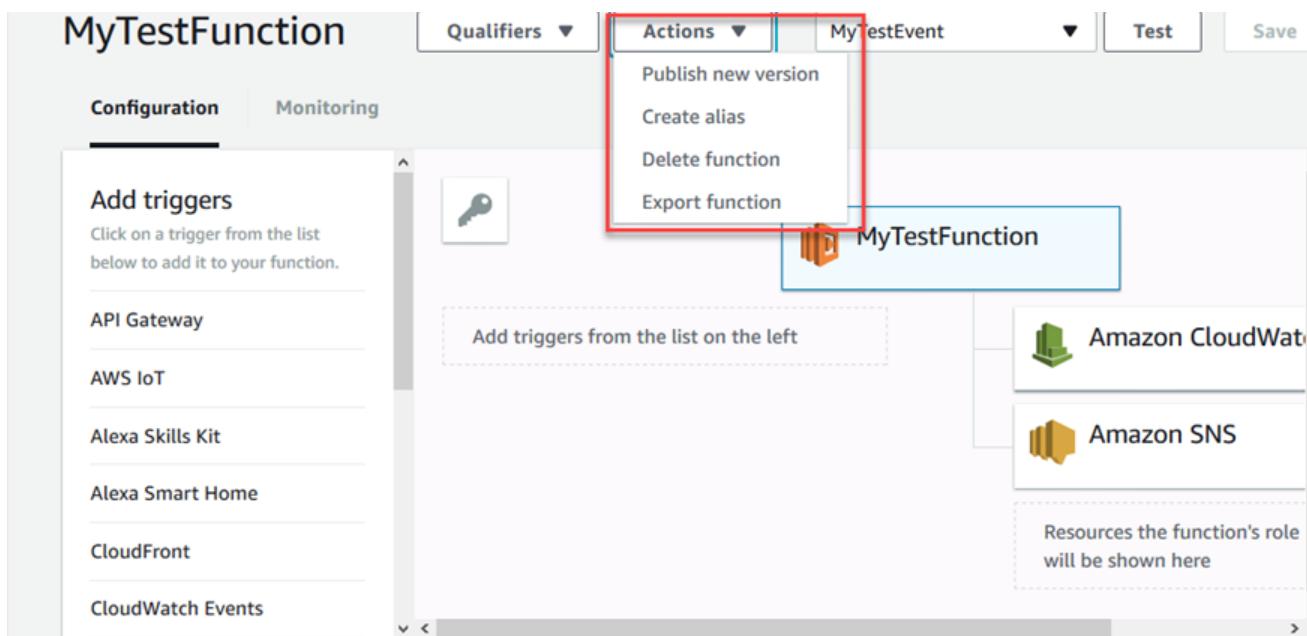
The expanded **Qualifiers** menu displays a **Versions** and **Aliases** tab, as shown in the following screen shot. In the **Versions** pane, you can see a list of versions for the selected function. If you have not previously published a version for the selected function, the **Versions** pane lists only the **\$LATEST** version, as shown following.



Choose the **Aliases** tab to see a list of aliases for the function. Initially, you won't have any aliases, as shown following.



Now you can publish a version or create aliases for the selected Lambda function by using the **Actions** menu.



To learn about versioning and aliases, see [AWS Lambda Function Versioning and Aliases \(p. 47\)](#).

Traffic Shifting Using Aliases

By default, an alias points to a single Lambda function version. When the alias is updated to point to a different function version, incoming request traffic in turn instantly points to the updated version. This exposes that alias to any potential instabilities introduced by the new version. To minimize this impact, you can implement the `routing-config` parameter of the Lambda alias that allows you to point to two different versions of the Lambda function and dictate what percentage of incoming traffic is sent to each version.

For example, you can specify that only 2 percent of incoming traffic is routed to the new version while you analyze its readiness for a production environment, while the remaining 98 percent is routed to the original version. As the new version matures, you can gradually update the ratio as necessary until you have determined the new version is stable. You can then update the alias to route all traffic to the new version.

You can point an alias to a maximum of two Lambda function versions. In addition:

- Both versions must have the same IAM execution role.
- Both versions must have the same [AWS Lambda Function Dead Letter Queues \(p. 88\)](#) configuration, or no DLQ configuration.
- When pointing an alias to more than one version, the alias cannot point to `$LATEST`.

Traffic Shifting Using an Alias (CLI)

To configure an alias to shift traffic between two function versions based on weights by using the [CreateAlias \(p. 347\)](#) operation, you need to configure the `routing-config` parameter. The example following points an alias to two different Lambda function versions, with version 2 receiving 2 percent of the invocation traffic and the remaining 98 percent invoking version 1.

```
aws lambda create-alias --name alias name --function-name function-name \ --function-version 1
```

```
--routing-config AdditionalVersionWeights={"2":0.02}
```

You can update the percentage of incoming traffic to your new version (version 2) by using the [UpdateAlias \(p. 448\)](#) operation. For example, you can boost the invocation traffic to your new version to 5 percent, as shown following.

```
aws lambda update-alias --name alias name --function-name function-name \  
--routing-config AdditionalVersionWeights={"2":0.05}
```

To route all traffic to version 2, again use the `UpdateAlias` operation to change the `function-version` property to point to version 2. In the same command, reset the routing configuration.

```
aws lambda update-alias --name alias name --function-name function-name \  
--function-version 2 --routing-config AdditionalVersionWeights={}
```

Traffic Shifting Using an Alias (Console)

You can configure traffic shifting with an alias by using the Lambda console as described below:

1. Open your Lambda function and verify that you have at least two previously published versions. Otherwise, you can go to [Introduction to AWS Lambda Versioning \(p. 48\)](#) to learn more about versioning, and publish your first function version.
2. For **Actions**, choose **Create alias**.
3. In the **Create a new alias** window, specify a value for **Name***, optionally for **Description**, and for **Version*** of the Lambda function that the alias will point to. Here the version is 1.
4. Under **Additional version**, specify the following:
 - a. Specify a second Lambda function version.
 - b. Type a weight value for the function. *Weight* is the percentage of traffic that is assigned to that version when the alias is invoked. The first version receives the residual weight. For example, if you specify 10 percent to **Additional version**, the first version automatically is assigned 90 percent.
5. Choose **Create**.

Determining Which Version Has Been Invoked

When your alias is shifting traffic between two function versions, there are two ways to determine which Lambda function version has been invoked:

1. **CloudWatch Logs** – Lambda automatically emits a `START` log entry that contains the invoked version ID to CloudWatch Logs for every function invocation. An example follows.

```
19:44:37 START RequestId: request id Version: $version
```

Lambda uses the `Executed Version` dimension to filter the metric data by the executed version. This only applies to alias invocations. For more information, see [AWS Lambda CloudWatch Dimensions \(p. 232\)](#).

2. **Response payload (synchronous invocations)** – Responses to synchronous function invocations include an `x-amz-executed-version` header to indicate which function version has been invoked.

AWS Lambda Layers

You can configure your Lambda function to pull in additional code and content in the form of layers. A layer is a ZIP archive that contains libraries, a [custom runtime \(p. 106\)](#), or other dependencies. With layers, you can use libraries in your function without needing to include them in your deployment package.

Layers let you keep your deployment package small, which makes development easier. You can avoid errors that can occur when you install and package dependencies with your function code. For Node.js, Python, and Ruby functions, you can [develop your function code in the Lambda console \(p. 24\)](#) as long as you keep your deployment package under 3 MB.

Note

A function can use up to 5 layers at a time. The total unzipped size of the function and all layers can't exceed the unzipped deployment package size limit of 250 MB. For more information, see [AWS Lambda Limits \(p. 7\)](#).

You can create layers, or use layers published by AWS and other AWS customers. Layers support [resource-based policies \(p. 68\)](#) for granting layer usage permissions to specific AWS accounts, [AWS Organizations](#), or all accounts.

Layers are extracted to the /opt directory in the function execution environment. Each runtime looks for libraries in a different location under /opt, depending on the language. [Structure your layer \(p. 67\)](#) so that function code can access libraries without additional configuration.

You can also use AWS Serverless Application Model (AWS SAM) to manage layers and your function's layer configuration. For instructions, see [Declaring Serverless Resources](#) in the *AWS Serverless Application Model Developer Guide*.

Sections

- [Configuring a Function to Use Layers \(p. 64\)](#)
- [Managing Layers \(p. 65\)](#)
- [Including Library Dependencies in a Layer \(p. 67\)](#)
- [Layer Permissions \(p. 68\)](#)

Configuring a Function to Use Layers

You can specify up to 5 layers in your function's configuration, during or after function creation. You choose a specific version of a layer to use. If you want to use a different version later, update your function's configuration.

To add layers to your function, use the `update-function-configuration` command. The following example adds two layers: one from the same account as the function, and one from a different account.

```
$ aws lambda update-function-configuration --function-name my-function \
--layers arn:aws:lambda:us-east-2:123456789012:layer:my-layer:3
 \
arn:aws:lambda:us-east-2:210987654321:layer:their-layer:2
{
    "FunctionName": "test-layers",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "Runtime": "nodejs8.10",
    "Role": "arn:aws:iam::123456789012:role/service-role/lambda-role",
    "Handler": "index.handler",
    "CodeSize": 402,
    "Description": "",
```

```

    "Timeout": 5,
    "MemorySize": 128,
    "LastModified": "2018-11-14T22:47:04.542+0000",
    "CodeSha256": "kDHAEY62Ni3OovMwVO8tNvgbRoRa6IOOKqShm7bSWF4=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "Active"
    },
    "RevisionId": "81cc64f5-5772-449a-b63e-12330476bcc4",
    "Layers": [
        {
            "Arn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer:3",
            "CodeSize": 169
        },
        {
            "Arn": "arn:aws:lambda:us-east-2:210987654321:layer:their-layer:2",
            "CodeSize": 169
        }
    ]
}

```

You must specify the version of each layer to use by providing the full ARN of the layer version. When you add layers to a function that already has layers, the previous list is overwritten by the new one. Include all layers every time you update the layer configuration. To remove all layers, specify an empty list.

```
$ aws lambda update-function-configuration --function-name my-function --layers []
```

Your function can access the content of the layer during execution in the /opt directory. Layers are applied in the order that's specified, merging any folders with the same name. If the same file appears in multiple layers, the version in the last applied layer is used.

The creator of a layer can delete the version of the layer that you're using. When this happens, your function continues to run as though the layer version still existed. However, when you update the layer configuration, you must remove the reference to the deleted version.

Managing Layers

To create a layer, use the `publish-layer-version` command with a name, description, ZIP archive, and a list of [runtimes \(p. 102\)](#) that are compatible with the layer. The list of runtimes is optional, but it makes the layer easier to discover.

```

$ aws lambda publish-layer-version --layer-name my-layer --description "My layer" --
license-info "MIT" \
--content S3Bucket=lambda-layers-us-east-2-123456789012,S3Key=layer.zip --compatible-
runtimes python3.6 python3.7
{
    "Content": {
        "Location": "https://awslambda-us-east-2-layers.s3.us-east-2.amazonaws.com/
snapshots/123456789012/my-layer-4aaa2fbb-ff77-4b0a-ad92-5b78a716a96a?
versionId=27iWyA73cCAYqyH...",
        "CodeSha256": "tv9jJ0+rPbXUUXuRKi7CwHzKtLDkDRJLB3cC3Z/ouXo=",
        "CodeSize": 169
    },
    "LayerArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer",
    "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer:1",
    "Description": "My layer",
    "CreatedDate": "2018-11-14T23:03:52.894+0000",
    "Version": 1,
    "LicenseInfo": "MIT",
    "CompatibleRuntimes": [

```

```

        "python3.6",
        "python3.7"
    ]
}
```

Each time you call `publish-layer-version`, you create a new version. Functions that use the layer refer directly to a layer version. You can [configure permissions \(p. 68\)](#) on an existing layer version, but to make any other changes, you must create a new version.

To find layers that are compatible with your function's runtime, use the `list-layers` command.

```
$ aws lambda list-layers --compatible-runtime python3.7
{
    "Layers": [
        {
            "LayerName": "my-layer",
            "LayerArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer",
            "LatestMatchingVersion": {
                "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer:2",
                "Version": 2,
                "Description": "My layer",
                "CreatedDate": "2018-11-15T00:37:46.592+0000",
                "CompatibleRuntimes": [
                    "python3.6",
                    "python3.7"
                ]
            }
        }
    ]
}
```

You can omit the runtime option to list all layers. The details in the response reflect the latest version of the layer. See all the versions of a layer with `list-layer-versions`. To see more information about a version, use `get-layer-version`.

```
$ aws lambda get-layer-version --layer-name my-layer --version-number 2
{
    "Content": {
        "Location": "https://awslambda-us-east-2-layers.s3.us-east-2.amazonaws.com/snapshots/123456789012/my-layer-91e9ea6e-492d-4100-97d5-a4388d442f3f?versionId=GmvPV.3090EpkfN...",
        "CodeSha256": "tv9jJO+rPbXUUXuRKi7CwHzKtLDkDRJLB3cC3Z/ouXo=",
        "CodeSize": 169
    },
    "LayerArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer",
    "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer:2",
    "Description": "My layer",
    "CreatedDate": "2018-11-15T00:37:46.592+0000",
    "Version": 2,
    "CompatibleRuntimes": [
        "python3.6",
        "python3.7"
    ]
}
```

The link in the response lets you download the layer archive and is valid for 10 minutes. To delete a layer version, use the `delete-layer-version` command.

```
$ aws lambda delete-layer-version --layer-name my-layer --version-number 1
```

When you delete a layer version, you can no longer configure functions to use it. However, any function that already uses the version continues to have access to it. Version numbers are never re-used for a layer name.

Including Library Dependencies in a Layer

You can move runtime dependencies out of your function code by placing them in a layer. Lambda runtimes include paths in the /opt directory to ensure that your function code has access to libraries that are included in layers.

To include libraries in a layer, place them in one of the folders supported by your runtime.

- **Node.js** – nodejs/node_modules, nodejs/node8/node_modules (NODE_PATH)

Example AWS X-Ray SDK for Node.js

```
xray-sdk.zip  
# nodejs/node_modules/aws-xray-sdk
```

- **Python** – python, python/lib/python3.7/site-packages (site directories)

Example Pillow

```
pillow.zip  
# python/PIL  
# python/Pillow-5.3.0.dist-info
```

- **Java** – java/lib (classpath)

Example Jackson

```
jackson.zip  
# java/lib/jackson-core-2.2.3.jar
```

- **Ruby** – ruby/gems/2.5.0 (GEM_PATH), ruby/lib (RUBY_LIB)

Example JSON

```
json.zip  
# ruby/gems/2.5.0/  
| build_info  
| cache  
| doc  
| extensions  
| gems  
| # json-2.1.0  
# specifications  
# json-2.1.0.gemspec
```

- **All** – bin (PATH), lib (LD_LIBRARY_PATH)

Example JQ

```
jq.zip  
# bin/jq
```

For more information about path settings in the Lambda execution environment, see [Environment Variables Available to Lambda Functions \(p. 103\)](#).

Layer Permissions

Layer usage permissions are managed on the resource. To configure a function with a layer, you need permission to call `GetLayerVersion` on the layer version. For functions in your account, you can get this permission from your [user policy \(p. 13\)](#) or from the function's [resource-based policy \(p. 10\)](#). To use a layer in another account, you need permission on your user policy, and the owner of the other account must grant your account permission with a resource-based policy.

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

```
$ aws lambda add-layer-version-permission --layer-name xray-sdk-nodejs --statement-id
xaccount \
--action lambda:GetLayerVersion --principal 210987654321 --version-number 1 --output text
e210ffdc-e901-43b0-824b-5fc0dd26d16 {"Sid":"xaccount","Effect":"Allow","Principal": {
"AWS":"arn:aws:iam::210987654321:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:lambda:us-east-2:123456789012:layer:xray-sdk-nodejs:1"}
```

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

For more examples, see [Granting Layer Access to Other Accounts \(p. 12\)](#).

Configuring a Lambda Function to Access Resources in an Amazon VPC

You can configure a function to connect to a virtual private cloud (VPC) in your account. Use Amazon Virtual Private Cloud (Amazon VPC) to create a private network for resources such as databases, cache instances, or internal services. Connect your function to the VPC to access private resources during execution.

AWS Lambda runs your function code securely within a VPC by default. However, to enable your Lambda function to access resources inside your private VPC, you must provide additional VPC-specific configuration information that includes VPC subnet IDs and security group IDs. AWS Lambda uses this information to set up elastic network interfaces ([ENIs](#)) that enable your function to connect securely to other resources within your private VPC.

Lambda functions cannot connect directly to a VPC with [dedicated instance tenancy](#). To connect to resources in a dedicated VPC, [peer it to a second VPC with default tenancy](#).

Configuring a Lambda Function for Amazon VPC Access

You add VPC information to your Lambda function configuration using the `VpcConfig` parameter, either at the time you create a Lambda function (see [CreateFunction \(p. 355\)](#)), or you can add it to the existing Lambda function configuration (see [UpdateFunctionConfiguration \(p. 463\)](#)). Following are AWS CLI examples:

- The `create-function` CLI command specifies the `--vpc-config` parameter to provide VPC information at the time you create a Lambda function.

```
$ aws lambda create-function \
--function-name ExampleFunction \
--runtime go1.x \
--role execution-role-arn \
--zip-file fileb://path/app.zip \
--handler app.handler \
--vpc-config SubnetIds=comma-separated-vpc-subnet-ids,SecurityGroupIds=comma-separated-security-group-ids \
--memory-size 1024
```

Note

The Lambda function execution role must have permissions to create, describe and delete ENIs. AWS Lambda provides a permissions policy, `AWSLambdaVPCAccessExecutionRole`, with permissions for the necessary EC2 actions (`ec2:CreateNetworkInterface`, `ec2:DescribeNetworkInterfaces`, and `ec2:DeleteNetworkInterface`) that you can use when creating a role. You can review the policy in the IAM console. Do not delete this role immediately after your Lambda function execution. There is a delay between the time your Lambda function executes and ENI deletion. If you do delete the role immediately after function execution, you are responsible for deleting the ENIs.

- The `update-function-configuration` CLI command specifies the `--vpc-config` parameter to add VPC information to an existing Lambda function configuration.

```
$ aws lambda update-function-configuration \
--function-name ExampleFunction \
--vpc-config SubnetIds=comma-separated-vpc-subnet-ids,SecurityGroupIds=security-group-ids
```

To remove VPC-related information from your Lambda function configuration, use the `UpdateFunctionConfiguration` API by providing an empty list of subnet IDs and security group IDs as shown in the following example CLI command.

```
$ aws lambda update-function-configuration \
--function-name ExampleFunction \
--vpc-config SubnetIds=[],SecurityGroupIds=[ ]
```

Note the following additional considerations:

- When you add VPC configuration to a Lambda function, it can only access resources in that VPC. If a Lambda function needs to access both VPC resources and the public Internet, the VPC needs to have a Network Address Translation (NAT) instance inside the VPC.
- When a Lambda function is configured to run within a VPC, it incurs an additional ENI start-up penalty. This means address resolution may be delayed when trying to connect to network resources.

Internet Access for Lambda Functions

AWS Lambda uses the VPC information you provide to set up `ENIs` that allow your Lambda function to access VPC resources. Each ENI is assigned a private IP address from the IP address range within the Subnets you specify, but is not assigned any public IP addresses. Therefore, if your Lambda function requires Internet access (for example, to access AWS services that don't have VPC endpoints), you can configure a NAT instance inside your VPC or you can use the Amazon VPC NAT gateway. For more information, see [NAT Gateways](#) in the *Amazon VPC User Guide*. You cannot use an Internet gateway attached to your VPC, since that requires the ENI to have public IP addresses.

If your Lambda function needs Internet access, do not attach it to a public subnet or to a private subnet without Internet access. Instead, attach it only to private subnets with Internet access through a NAT instance or an Amazon VPC NAT gateway.

Guidelines for Setting Up VPC-Enabled Lambda Functions

Your Lambda function automatically scales based on the number of events it processes. The following are general guidelines for setting up VPC-enabled Lambda functions to support the scaling behavior.

- If your Lambda function accesses a VPC, you must make sure that your VPC has sufficient ENI capacity to support the scale requirements of your Lambda function. You can use the following formula to approximately determine the ENI requirements.

```
Projected peak concurrent executions * (Memory in GB / 3GB)
```

Where:

- **Projected peak concurrent execution** – Use the information in [Managing Concurrency \(p. 37\)](#) to determine this value.
- **Memory** – The amount of memory you configured for your Lambda function.
- The subnets you specify should have sufficient available IP addresses to match the number of ENIs.

We also recommend that you specify at least one subnet in each Availability Zone in your Lambda function configuration. By specifying subnets in each of the Availability Zones, your Lambda function can run in another Availability Zone if one goes down or runs out of IP addresses.

If your VPC does not have sufficient ENIs or subnet IPs, your Lambda function will not scale as requests increase, and you will see an increase in invocation errors with EC2 error types like `EC2ThrottledException`. For asynchronous invocation, if you see an increase in errors without corresponding CloudWatch Logs, invoke the Lambda function synchronously in the console to get the error responses.

Tutorial: Configuring a Lambda Function to Access Amazon ElastiCache in an Amazon VPC

In this tutorial, you do the following:

- Create an Amazon ElastiCache cluster in your default Amazon Virtual Private Cloud. For more information about Amazon ElastiCache, see [Amazon ElastiCache](#).
- Create a Lambda function to access the ElastiCache cluster. When you create the Lambda function, you provide subnet IDs in your Amazon VPC and a VPC security group to allow the Lambda function to access resources in your VPC. For illustration in this tutorial, the Lambda function generates a UUID, writes it to the cache, and retrieves it from the cache.
- Invoke the Lambda function and verify that it accessed the ElastiCache cluster in your VPC.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Getting Started with AWS Lambda \(p. 3\)](#) to create your first Lambda function.

To follow the procedures in this guide, you will need a command line terminal or shell to run commands. Commands are shown in listings preceded by a prompt symbol (\$) and the name of the current directory, when appropriate:

```
~/lambda-project$ this is a command  
this is output
```

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

On Linux and macOS, use your preferred shell and package manager. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the Execution Role

Create the [execution role \(p. 9\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity** – Lambda.
 - **Permissions** – **AWSLambdaVPCAccessExecutionRole**.
 - **Role name** – **lambda-vpc-role**.

The **AWSLambdaVPCAccessExecutionRole** has the permissions that the function needs to manage network connections to a VPC.

Create an ElastiCache Cluster

Create an ElastiCache cluster in your default VPC.

1. Run the following AWS CLI command to create a Memcached cluster.

```
$ aws elasticache create-cache-cluster --cache-cluster-id ClusterForLambdaTest \  
--cache-node-type cache.m3.medium --engine memcached --num-cache-nodes 1 \  
--security-group-ids your-default-vpc-security-group
```

You can look up the default VPC security group in the VPC console under **Security Groups**. Your example Lambda function will add and retrieve an item from this cluster.

2. Write down the configuration endpoint for the cache cluster that you launched. You can get this from the Amazon ElastiCache console. You will specify this value in your Lambda function code in the next section.

Create a Deployment Package

The following example Python code reads and writes an item to your ElastiCache cluster.

Example app.py

```
from __future__ import print_function  
import time  
import uuid
```

```

import sys
import socket
import elasticache_auto_discovery
from pymemcache.client.hash import HashClient

#elasticache settings
elasticache_config_endpoint = "your-elasticacluster-endpoint:port"
nodes = elasticache_auto_discovery.discover(elasticache_config_endpoint)
nodes = map(lambda x: (x[1], int(x[2])), nodes)
memcache_client = HashClient(nodes)

def handler(event, context):
    """
    This function puts into memcache and get from it.
    Memcache is hosted using elasticache
    """

    #Create a random UUID... this will be the sample element we add to the cache.
    uuid_inserted = str(uuid.uuid4().hex)
    #Put the UUID to the cache.
    memcache_client.set('uuid', uuid_inserted)
    #Get item (UUID) from the cache.
    uuid_obtained = memcache_client.get('uuid')
    if uuid_obtained.decode("utf-8") == uuid_inserted:
        # this print should go to the CloudWatch Logs and Lambda console.
        print ("Success: Fetched value %s from memcache" %(uuid_inserted))
    else:
        raise Exception("Value is not the same as we put :(. Expected %s got %s"
        %(uuid_inserted, uuid_obtained))

    return "Fetched value from memcache: " + str(uuid_obtained.decode("utf-8"))

```

Dependencies

- [pymemcache](#) – The Lambda function code uses this library to create a HashClient object to set and get items from memcache.
- [elasticache-auto-discovery](#) – The Lambda function uses this library to get the nodes in your Amazon ElastiCache cluster.

Install dependencies with Pip and create a deployment package. For instructions, see [AWS Lambda Deployment Package in Python \(p. 251\)](#).

Create the Lambda Function

Create the Lambda function with the `create-function` command.

```
$ aws lambda create-function --function-name AccessMemCache --timeout 30 --memory-size 1024 \
\--zip-file fileb://function.zip --handler app.handler --runtime python3.7 \
\--role execution-role-arn \
\--vpc-config SubnetIds=comma-separated-vpc-subnet-ids,SecurityGroupIds=default-security-group-id
```

You can find the subnet IDs and the default security group ID of your VPC from the VPC console.

Test the Lambda Function

In this step, you invoke the Lambda function manually using the `invoke` command. When the Lambda function executes, it generates a UUID and writes it to the ElastiCache cluster that you specified in your Lambda code. The Lambda function then retrieves the item from the cache.

1. Invoke the Lambda function with the `invoke` command.

```
$ aws lambda invoke --function-name AccessMemCache output.txt
```

2. Verify that the Lambda function executed successfully as follows:

- Review the `output.txt` file.
- Review the results in the AWS Lambda console.
- Verify the results in CloudWatch Logs.

Now that you have created a Lambda function that accesses an ElastiCache cluster in your VPC, you can have the function invoked in response to events. For information about configuring event sources and examples, see [Using AWS Lambda with Other Services \(p. 132\)](#).

Tutorial: Configuring a Lambda Function to Access Amazon RDS in an Amazon VPC

In this tutorial, you do the following:

- Launch an Amazon RDS MySQL database engine instance in your default Amazon VPC. In the MySQL instance, you create a database (ExampleDB) with a sample table (Employee) in it. For more information about Amazon RDS, see [Amazon RDS](#).
- Create a Lambda function to access the ExampleDB database, create a table (Employee), add a few records, and retrieve the records from the table.
- Invoke the Lambda function and verify the query results. This is how you verify that your Lambda function was able to access the RDS MySQL instance in the VPC.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Getting Started with AWS Lambda \(p. 3\)](#) to create your first Lambda function.

To follow the procedures in this guide, you will need a command line terminal or shell to run commands. Commands are shown in listings preceded by a prompt symbol (\$) and the name of the current directory, when appropriate:

```
~/lambda-project$ this is a command  
this is output
```

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

On Linux and macOS, use your preferred shell and package manager. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the Execution Role

Create the [execution role \(p. 9\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.

3. Create a role with the following properties.

- **Trusted entity** – Lambda.
- **Permissions** – **AWSLambdaVPCAccessExecutionRole**.
- **Role name** – **lambda-vpc-role**.

The **AWSLambdaVPCAccessExecutionRole** has the permissions that the function needs to manage network connections to a VPC.

Create an Amazon RDS Database Instance

In this tutorial, the example Lambda function creates a table (`Employee`), inserts a few records, and then retrieves the records. The table that the Lambda function creates has the following schema:

```
Employee(EmpID, Name)
```

Where `EmpID` is the primary key. Now, you need to add a few records to this table.

First, you launch an RDS MySQL instance in your default VPC with `ExampleDB` database. If you already have an RDS MySQL instance running in your default VPC, skip this step.

You can launch an RDS MySQL instance using one of the following methods:

- Follow the instructions at [Creating a MySQL DB Instance and Connecting to a Database on a MySQL DB Instance](#) in the *Amazon RDS User Guide*.
- Use the following AWS CLI command:

```
$ aws rds create-db-instance --db-name ExampleDB --engine MySQL \
--db-instance-identifier MySQLForLambdaTest --backup-retention-period 3 \
--db-instance-class db.t2.micro --allocated-storage 5 --no-publicly-accessible \
--master-username username --master-user-password password
```

Write down the database name, user name, and password. You also need the host address (endpoint) of the DB instance, which you can get from the RDS console. You might need to wait until the instance status is available and the Endpoint value appears in the console.

Create a Deployment Package

The following example Python code runs a `SELECT` query against the `Employee` table in the MySQL RDS instance that you created in the VPC. The code creates a table in the `ExampleDB` database, adds sample records, and retrieves those records.

Example app.py

```
import sys
import logging
import rds_config
import pymysql
#rds settings
rds_host = "rds-instance-endpoint"
name = rds_config.db_username
password = rds_config.db_password
db_name = rds_config.db_name

logger = logging.getLogger()
logger.setLevel(logging.INFO)
```

```
try:
    conn = pymysql.connect(rds_host, user=name, passwd=password, db=db_name,
    connect_timeout=5)
except:
    logger.error("ERROR: Unexpected error: Could not connect to MySQL instance.")
    sys.exit()

logger.info("SUCCESS: Connection to RDS MySQL instance succeeded")
def handler(event, context):
    """
    This function fetches content from MySQL RDS instance
    """

    item_count = 0

    with conn.cursor() as cur:
        cur.execute("create table Employee3 ( EmpID  int NOT NULL, Name varchar(255) NOT
NULL, PRIMARY KEY (EmpID))")
        cur.execute('insert into Employee3 (EmpID, Name) values(1, "Joe")')
        cur.execute('insert into Employee3 (EmpID, Name) values(2, "Bob")')
        cur.execute('insert into Employee3 (EmpID, Name) values(3, "Mary")')
        conn.commit()
        cur.execute("select * from Employee3")
        for row in cur:
            item_count += 1
            logger.info(row)
            #print(row)
        conn.commit()

    return "Added %d items from RDS MySQL table" %(item_count)
```

Executing `pymysql.connect()` outside of the handler allows your function to re-use the database connection for better performance.

A second file contains connection information for the function.

Example rds_config.py

```
#config file containing credentials for RDS MySQL instance
db_username = "username"
db_password = "password"
db_name = "ExampleDB"
```

Dependencies

- `pymysql` – The Lambda function code uses this library to access your MySQL instance (see [PyMySQL](#)).

Install dependencies with Pip and create a deployment package. For instructions, see [AWS Lambda Deployment Package in Python \(p. 251\)](#).

Create the Lambda Function

Create the Lambda function with the `create-function` command.

```
$ aws lambda create-function --function-name CreateTableAddRecordsAndRead --runtime
python3.7 \
--zip-file fileb://app.zip --handler app.handler \
--role execution-role-arn \
--vpc-config SubnetIds=comma-separated-subnet-ids,SecurityGroupIds=default-vpc-security-
group-id
```

Test the Lambda Function

In this step, you invoke the Lambda function manually using the `invoke` command. When the Lambda function executes, it runs the `SELECT` query against the `Employee` table in the RDS MySQL instance and prints the results, which also go to the CloudWatch Logs.

1. Invoke the Lambda function with the `invoke` command.

```
$ aws lambda invoke --function-name CreateTableAddRecordsAndRead output.txt
```

2. Verify that the Lambda function executed successfully as follows:

- Review the `output.txt` file.
- Review the results in the AWS Lambda console.
- Verify the results in CloudWatch Logs.

Now that you have created a Lambda function that accesses a database in your VPC, you can have the function invoked in response to events. For information about configuring event sources and examples, see [Using AWS Lambda with Other Services \(p. 132\)](#).

Tagging Lambda Functions

Lambda functions can span multiple applications across separate regions. To simplify the process of tracking the frequency and cost of each function invocation, you can use tags. Tags are key-value pairs that you attach to AWS resources to better organize them. They are particularly useful when you have many resources of the same type, which in the case of AWS Lambda, is a function. By using tags, customers with hundreds of Lambda functions can easily access and analyze a specific set by filtering on those that contain the same tag. Two of the key advantages of tagging your Lambda functions are:

- **Grouping and Filtering:** By applying tags, you can use the Lambda console or CLI to isolate a list of Lambda functions contained within a specific application or billing department. For more information, see [Filtering on Tagged Lambda Functions \(p. 78\)](#).
- **Cost allocation:** Because Lambda's support for tagging is integrated with AWS Billing, you can break down bills into dynamic categories and map functions to specific cost centers. For example, if you tag all Lambda functions with a "Department" key, then all AWS Lambda costs can be broken down by department. You can then provide an individual department value, such "Department 1" or "Department 2" to direct the function invocation cost to the appropriate cost center. Cost allocation is surfaced via detailed billing reports, making it easier for you to categorize and track your AWS costs.

Topics

- [Tagging Lambda Functions for Billing \(p. 76\)](#)
- [Applying Tags to Lambda Functions Using the Console \(p. 77\)](#)
- [Applying Tags to Lambda Functions Using the CLI \(p. 77\)](#)
- [Filtering on Tagged Lambda Functions \(p. 78\)](#)
- [Tag Restrictions \(p. 79\)](#)

Tagging Lambda Functions for Billing

You can use tags to organize your AWS bill to reflect your own cost structure. To do this, you can add tag keys whose values will be included in the cost allocation report. For more information about setting up a

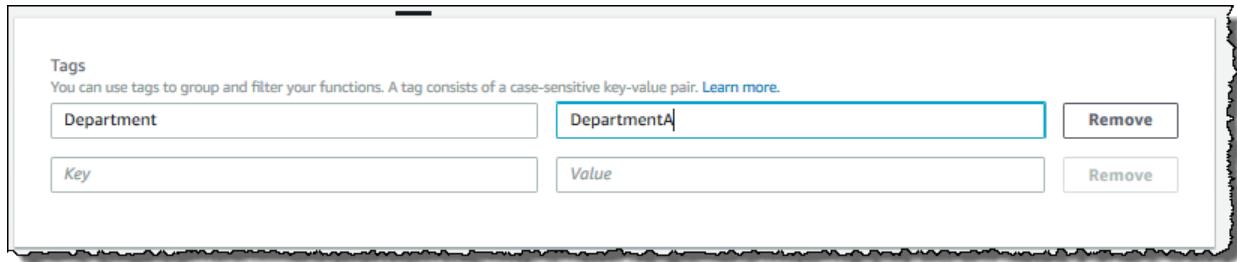
cost allocation report that includes the tag keys you select to be included as line items in the report, see [The Monthly Cost Allocation Report](#) in *About AWS Account Billing*.

To see the cost of your combined resources, you can organize your billing information based on functions that have the same tag key values. For example, you can tag several Lambda functions with a specific application name, and then organize your billing information to see the total cost of that application across several services. For more information, see [Using Cost Allocation Tags](#) in the *AWS Billing and Cost Management User Guide*.

In AWS Lambda the only resource that can be tagged is a function. You cannot tag an alias or a specific function version. Any invocation of a function's alias or version will be billed as an invocation of the original function.

Applying Tags to Lambda Functions Using the Console

You can add tags to your function under the **Tags** section in the **configuration** tab.



To remove tags from an existing function, open the function, choose the **Tags** section and then choose the **Remove** button next to key-value pair.



Applying Tags to Lambda Functions Using the CLI

When you create a new Lambda function, you can include tags with the `--tags` option.

```
$ aws lambda create-function --function-name my-function
--handler index.js --runtime nodejs8.10 \
--role role-arn \
--tags "DEPARTMENT=Department A"
```

To add tags to an existing function, use the `tag-resource` command.

```
$ aws lambda tag-resource \
--resource function arn \
--tags "DEPARTMENT=Department A"
```

To remove tags, use the `untag-resource` command.

```
$ aws lambda untag-resource --resource function arn \
```

```
--tagkeys DEPARTMENT
```

Filtering on Tagged Lambda Functions

Once you have grouped your Lambda functions by using tags, you can leverage the filtering capabilities provided by the Lambda console or the AWS CLI to view them based on your specific requirements.

Filtering Lambda Functions Using the Console

The Lambda console contains a search field that allows you to filter the list of functions based on a specified set of function attributes, including **Tags**. Suppose you have two functions named **MyFunction** and **MyFunction2** that have a **Tags** key called **Department**. To view those functions, choose the search field and notice the automatic filtering that includes a list of the **Tags** keys:

Functions (25)			
<input type="text"/> Filter by tags and attributes or search by keyword			
	Function attributes	Runtime	Code size
<input type="checkbox"/>	Description		
<input type="checkbox"/>	Function name	mbda function.	Node.js 6.10
<input type="checkbox"/>	Runtime	mbda function.	Node.js 6.10
<input checked="" type="checkbox"/>	Tags	mbda function.	333 bytes
<input type="checkbox"/>	Department	mbda function.	Python 2.7
			360 bytes

Choose the **Department** key. Lambda will return any function that contains that key.

Now suppose that the key value of the **MyFunction** tag is "Department A" and the key value of **MyFunction2** is "Department B". You can narrow your search by choosing the value of the **Department** key, in this case **Department A**, as shown below.

Functions (25)			
<input type="text"/> tag:Department :			
	Function attributes	Runtime	Code size
<input type="checkbox"/>	(all values)		
<input type="checkbox"/>	(empty)		
<input checked="" type="checkbox"/>	DepartmentA	mbda function.	Node.js 6.10
<input type="checkbox"/>	DepartmentB	mbda function.	Node.js 6.10
			333 bytes
			333 bytes

This will return only **MyFunction**.

You can further narrow your search by including the other accepted **Function attributes**, including **Description**, **Function name** or **Runtime**.

Note

You are limited to a maximum of 50 tags per Lambda function. If you delete the Lambda function, the associated tags will also be deleted.

Filtering Lambda Functions Using the CLI

If you want to view the tags that are applied to a specific Lambda function, you can use either of the following Lambda API commands:

- [ListTags \(p. 421\)](#): You supply your Lambda function ARN (Amazon Resource Name) to view a list of the tags associated with this function:

```
$ aws lambda list-tags --resource function arn
```

- [GetFunction \(p. 382\)](#): You supply your Lambda function name to view a list of the tags associated with this function:

```
$ aws lambda get-function --function-name my-function
```

You can also use the AWS Tagging Service's [GetResources](#) API to filter your resources by tags. The GetResources API receives up to 10 filters, with each filter containing a tag key and up to 10 tag values. You provide GetResources with a 'ResourceType' to filter by specific resource types. For more information about the AWS Tagging Service, see [Working with Resource Groups](#).

Tag Restrictions

The following restrictions apply to tags:

- Maximum number of tags per resource—50
- Maximum key length—128 Unicode characters in UTF-8
- Maximum value length—256 Unicode characters in UTF-8
- Tag keys and values are case sensitive.
- Do not use the `aws :` prefix in your tag names or values because it is reserved for AWS use. You can't edit or delete tag names or values with this prefix. Tags with this prefix do not count against your tags per resource limit.
- If your tagging schema will be used across multiple services and resources, remember that other services may have restrictions on allowed characters. Generally allowed characters are: letters, spaces, and numbers representable in UTF-8, plus the following special characters: `+ - = . _ : / @.`

Invoking AWS Lambda Functions

When building applications on AWS Lambda the core components are Lambda functions and event sources. An *event source* is the AWS service or custom application that publishes events, and a *Lambda function* is the custom code that processes the events. To illustrate, consider the following scenarios:

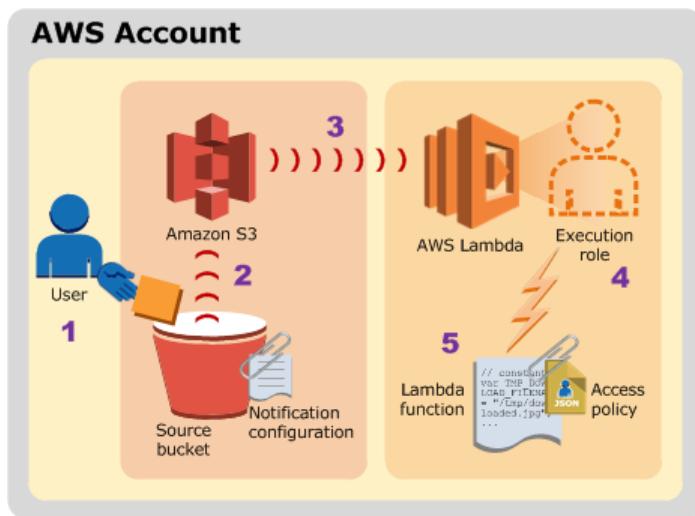
- **File processing** – Suppose you have a photo sharing application. People use your application to upload photos, and the application stores these user photos in an Amazon S3 bucket. Then, your application creates a thumbnail version of each user's photos and displays them on the user's profile page. In this scenario, you may choose to create a Lambda function that creates a thumbnail automatically. Amazon S3 is one of the supported AWS event sources that can publish *object-created events* and invoke your Lambda function. Your Lambda function code can read the photo object from the S3 bucket, create a thumbnail version, and then save it in another S3 bucket.
- **Data and analytics** – Suppose you are building an analytics application and storing raw data in a DynamoDB table. When you write, update, or delete items in a table, DynamoDB streams can publish item update events to a stream associated with the table. In this case, the event data provides the item key, event name (such as insert, update, and delete), and other relevant details. You can write a Lambda function to generate custom metrics by aggregating raw data.
- **Websites** – Suppose you are creating a website and you want to host the backend logic on Lambda. You can invoke your Lambda function over HTTP using Amazon API Gateway as the HTTP endpoint. Now, your web client can invoke the API, and then API Gateway can route the request to Lambda.
- **Mobile applications** – Suppose you have a custom mobile application that produces events. You can create a Lambda function to process events published by your custom application. For example, in this scenario you can configure a Lambda function to process the clicks within your custom mobile application.

AWS Lambda supports many AWS services as event sources. For more information, see [Using AWS Lambda with Other Services \(p. 132\)](#). When you configure these event sources to trigger a Lambda function, the Lambda function is invoked automatically when events occur. You define *event source mapping*, which is how you identify what events to track and which Lambda function to invoke.

The following are introductory examples of event sources and how the end-to-end experience works.

Example 1: Amazon S3 Pushes Events and Invokes a Lambda Function

Amazon S3 can publish events of different types, such as PUT, POST, COPY, and DELETE object events on a bucket. Using the bucket notification feature, you can configure an event source mapping that directs Amazon S3 to invoke a Lambda function when a specific type of event occurs, as shown in the following illustration.



The diagram illustrates the following sequence:

1. The user creates an object in a bucket.
2. Amazon S3 detects the object created event.
3. Amazon S3 invokes your Lambda function using the permissions provided by the [execution role \(p. 9\)](#).
4. AWS Lambda executes the Lambda function, specifying the event as a parameter.

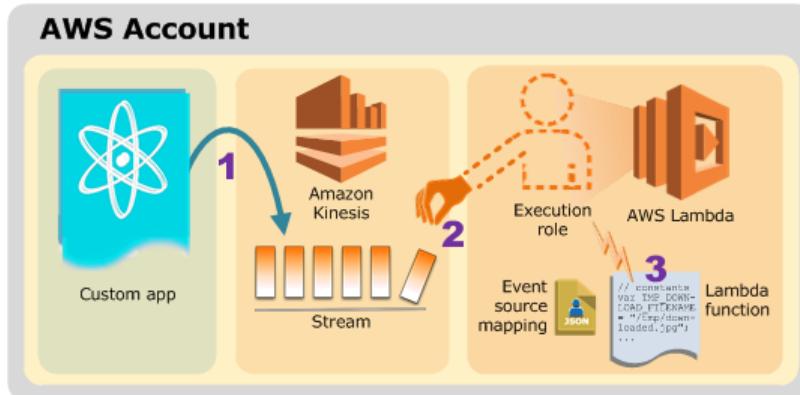
You configure Amazon S3 to invoke your function as a bucket notification action. To grant Amazon S3 permission to invoke the function, update the function's [resource-based policy \(p. 10\)](#).

Example 2: AWS Lambda Pulls Events from a Kinesis Stream and Invokes a Lambda Function

For poll-based event sources, AWS Lambda polls the source and then invokes the Lambda function when records are detected on that source.

- [CreateEventSourceMapping \(p. 351\)](#)
- [UpdateEventSourceMapping \(p. 452\)](#)

The following diagram shows how a custom application writes records to a Kinesis stream.



The diagram illustrates the following sequence:

1. The custom application writes records to a Kinesis stream.
2. AWS Lambda continuously polls the stream, and invokes the Lambda function when the service detects new records on the stream. AWS Lambda knows which stream to poll and which Lambda function to invoke based on the event source mapping you create in Lambda.
3. The Lambda function is invoked with the incoming event.

When working with stream-based event sources, you create event source mappings in AWS Lambda. Lambda reads items from the stream invokes the function synchronously. You don't need to grant Lambda permission to invoke the function, but it does need permission to read from the stream.

Invocation Types

AWS Lambda supports synchronous and asynchronous invocation of a Lambda function. You can control the invocation type only when you invoke a Lambda function (referred to as *on-demand invocation*). The following examples illustrate on-demand invocations:

- Your custom application invokes a Lambda function.
- You manually invoke a Lambda function (for example, using the AWS CLI) for testing purposes.

In both cases, you invoke your Lambda function using the [Invoke \(p. 400\)](#) operation, and you can specify the invocation type as synchronous or asynchronous.

When you use AWS services as a trigger, the invocation type is predetermined for each service. You have no control over the invocation type that these event sources use when they invoke your Lambda function.

For example, Amazon S3 always invokes a Lambda function asynchronously and Amazon Cognito always invokes a Lambda function synchronously. For poll-based AWS services (Amazon Kinesis, Amazon DynamoDB, Amazon Simple Queue Service), AWS Lambda polls the stream or message queue and invokes your Lambda function synchronously.

AWS Lambda Event Source Mapping

Lambda functions and event sources are the core components of AWS Lambda. An event source is the entity that publishes events, and a Lambda function is the custom code that processes the events. Supported event sources are the AWS services that can be preconfigured to work with AWS Lambda. The configuration is referred to as *event source mapping*, which maps an event source to a Lambda function. It enables automatic invocation of your Lambda function when events occur.

Each event source mapping identifies the type of events to publish and the Lambda function to invoke when events occur. The specific Lambda function then receives the event information as a parameter and your Lambda function code then processes the event.

You can also create custom applications to include AWS resource events and invoke a Lambda function. For more information, see [Using AWS Lambda with the AWS Command Line Interface \(p. 89\)](#)

You may be wondering—where do I keep the event mapping information? Do I keep it within the event source or within AWS Lambda? The following sections explain event source mapping for each of these event source categories. These sections also explain how the Lambda function is invoked and how you manage permissions to allow invocation of your Lambda function.

Topics

- [Event Source Mapping for AWS Services \(p. 83\)](#)
- [Event Source Mapping for AWS Poll-Based Services \(p. 84\)](#)
- [Event Source Mapping for Custom Applications \(p. 84\)](#)

Event Source Mapping for AWS Services

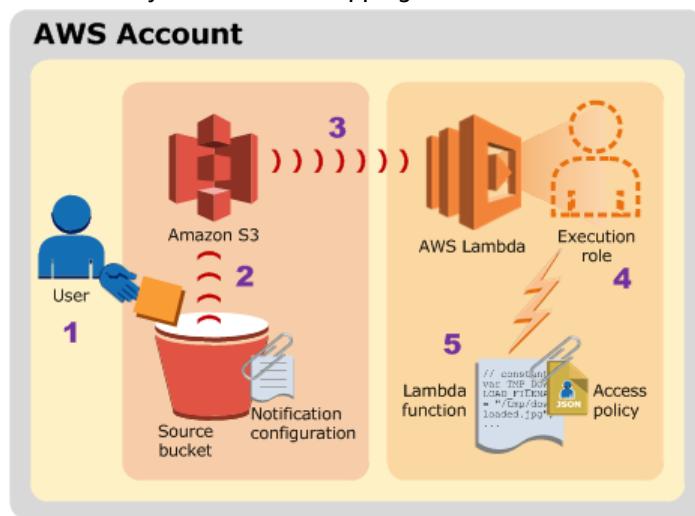
Except for the poll-based AWS services (Amazon Kinesis Data Streams and DynamoDB streams or Amazon SQS queues), other supported AWS services publish events and can also invoke your Lambda function (referred to as the *push model*). In the push model, note the following:

- Event source mappings are maintained within the event source. Relevant API support in the event sources enables you to create and manage event source mappings. For example, Amazon S3 provides the bucket notification configuration API. Using this API, you can configure an event source mapping that identifies the bucket events to publish and the Lambda function to invoke.
- Because the event sources invoke your Lambda function, you need to grant the event source the necessary permissions using a resource-based policy. For more information, see [Using Resource-based Policies for AWS Lambda \(p. 10\)](#).

The following example illustrates how this model works.

Example – Amazon S3 Pushes Events and Invokes a Lambda Function

Suppose that you want your AWS Lambda function invoked for each *object created* bucket event. You add the necessary event source mapping in the bucket notification configuration.



The diagram illustrates the flow:

1. The user creates an object in a bucket.
2. Amazon S3 detects the object created event.
3. Amazon S3 invokes your Lambda function according to the event source mapping described in the bucket notification configuration.
4. AWS Lambda verifies the permissions policy attached to the Lambda function to ensure that Amazon S3 has the necessary permissions. For more information on permissions policies, see [AWS Lambda Permissions \(p. 9\)](#)
5. Once AWS Lambda verifies the attached permissions policy, it executes the Lambda function. Remember that your Lambda function receives the event as a parameter.

Event Source Mapping for AWS Poll-Based Services

AWS Lambda supports the following poll-based services:

Services That Lambda Reads Events From

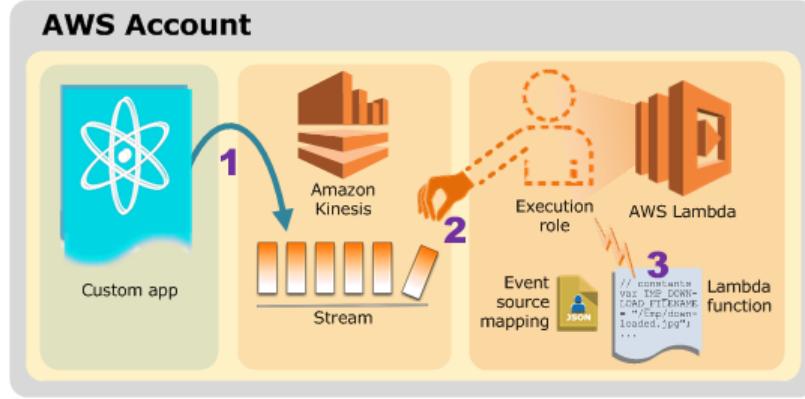
- [Amazon Kinesis \(p. 180\)](#)
- [Amazon DynamoDB \(p. 170\)](#)
- [Amazon Simple Queue Service \(p. 216\)](#)

Once you have configured the event source mapping, AWS Lambda polls the event source and invokes your Lambda function. The event source mappings are maintained within the AWS Lambda. AWS Lambda provides APIs to create and manage event source mappings. For more information, see [CreateEventSourceMapping \(p. 351\)](#).

AWS Lambda needs your permission to poll Kinesis and DynamoDB streams or Amazon SQS queues and read records. You grant these permissions via the [execution role \(p. 9\)](#), using the permissions policy associated with role that you specify when you create your Lambda function. AWS Lambda does not need any permissions to invoke your Lambda function.

The following diagram shows a custom application that writes records to a Kinesis stream and how AWS Lambda polls the stream. When AWS Lambda detects a new record on the stream, it invokes your Lambda function.

Suppose you have a custom application that writes records to a Kinesis stream. You want to invoke a Lambda function when new records are detected on the stream. You create a Lambda function and the necessary event source mapping in AWS Lambda.



The diagram illustrates the following sequence:

1. The custom application writes records to an Amazon Kinesis stream.
2. AWS Lambda continuously polls the stream and invokes the Lambda function once the service detects new records on the stream. AWS Lambda knows which stream to poll and which Lambda function to invoke based on the event source mapping you create in AWS Lambda.
3. AWS Lambda executes the Lambda function.

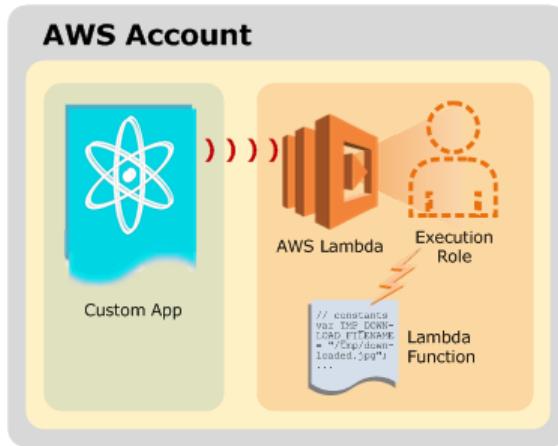
The example uses a Kinesis stream but the same applies when working with a DynamoDB stream.

Event Source Mapping for Custom Applications

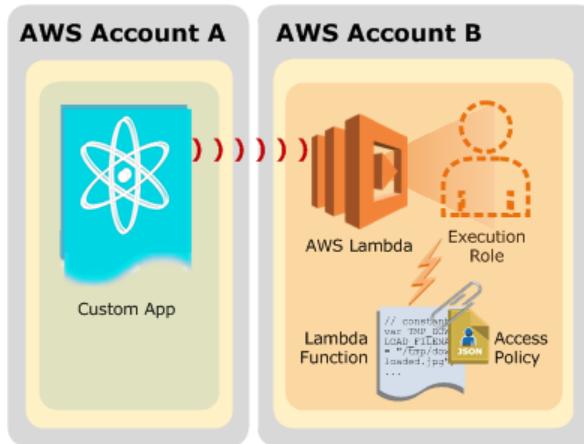
If you have custom applications that publish and process events, you can create a Lambda function to process these events. In this case, there is no preconfiguration required—you don't have to set up an

event source mapping. Instead, the event source uses the AWS Lambda Invoke API. If the application and Lambda function are owned by different AWS accounts, the AWS account that owns the Lambda function must allow cross-account permissions in the permissions policy associated with the Lambda function.

The following diagram shows how a custom application in your account can invoke a Lambda function. In this example, the custom application is using the same account credentials as the account that owns the Lambda function, and, therefore, does not require additional permissions to invoke the function.



In the following example, the user application and Lambda function are owned by different AWS accounts. In this case, the AWS account that owns the Lambda function must have cross-account permissions in the permissions policy associated with the Lambda function. For more information, see [AWS Lambda Permissions \(p. 9\)](#).



AWS Lambda Retry Behavior

Function invocation can result in an error for several reasons. Your code might raise an exception, time out, or run out of memory. The runtime executing your code might encounter an error and stop. You might run out of concurrency and be throttled.

When an error occurs, your code might have run completely, partially, or not at all. In most cases, the client or service that invokes your function retries if it encounters an error, so your code must be able to process the same event repeatedly without unwanted effects. If your function manages resources or writes to a database, you need to handle cases where the same request is made several times.

Lambda handles retries in the following manner, depending on the source of the invocation.

- **Event sources that aren't stream-based** – Some of these event sources are set up to invoke a Lambda function synchronously and others invoke it asynchronously. Accordingly, exceptions are handled as follows:
 - **Synchronous invocation** – Lambda includes the `FunctionError` field in the response body, with details about the error in the `X-Amz-Function-Error` header. The status code is 200 for function errors. Lambda only returns error status codes if there is an issue with the request, function, or permissions that prevents the handler from processing the event. See [Invoke Errors \(p. 402\)](#) for details.
 - **AWS service triggers (p. 132)** can retry depending on the service. If you invoke the Lambda function directly from your application, you can choose whether to retry or not.
 - **Asynchronous invocation** – Asynchronous events are queued before being used to invoke the Lambda function. If AWS Lambda is unable to fully process the event, it will automatically retry the invocation twice, with delays between retries. Configure a [dead letter queue \(p. 88\)](#) for your function to capture requests that fail all three attempts.
 - **Poll-based event sources that are stream-based** – These consist of Kinesis Data Streams or DynamoDB. When a Lambda function invocation fails, AWS Lambda attempts to process the erring batch of records until the time the data expires, which can be up to seven days.

The exception is treated as blocking, and AWS Lambda will not read any new records from the shard until the failed batch of records either expires or is processed successfully. This ensures that AWS Lambda processes the stream events in order.

- **Poll-based event sources that are not stream-based** – This consists of Amazon Simple Queue Service. If you configure an Amazon SQS queue as an event source, AWS Lambda will poll a batch of records in the queue and invoke your Lambda function. If the invocation fails or times out, every message in the batch will be returned to the queue, and each will be available for processing once the [Visibility Timeout](#) period expires. (Visibility timeouts are a period of time during which Amazon Simple Queue Service prevents other consumers from receiving and processing the message).

Once an invocation successfully processes a batch, each message in that batch will be removed from the queue. When a message is not successfully processed, it is either discarded or if you have configured an [Amazon SQS Dead Letter Queue](#), the failure information will be directed there for you to analyze.

If you don't require ordered processing of events, the advantage of using Amazon SQS queues is that AWS Lambda will continue to process new messages, regardless of a failed invocation of a previous message. In other words, processing of new messages will not be blocked.

For more information about invocation modes, see [AWS Lambda Event Source Mapping \(p. 82\)](#).

Understanding Scaling Behavior

Concurrent executions refers to the number of executions of your function code that are happening at any given time. You can estimate the concurrent execution count, but the concurrent execution count will differ depending on whether or not your Lambda function is processing events from a poll-based event source.

If you create a Lambda function to process events from event sources that aren't poll-based (for example, Lambda can process every event from other sources, like Amazon S3 or API Gateway), each published event is a unit of work, in parallel, up to your account limits. Therefore, the number of invocations these event sources make influences the concurrency. You can use the this formula to estimate the capacity used by your function:

```
invocations per second * average execution duration in seconds
```

For example, consider a Lambda function that processes Amazon S3 events. Suppose that the Lambda function takes on average three seconds and Amazon S3 publishes 10 events per second. Then, you will have 30 concurrent executions of your Lambda function.

The number of concurrent executions for poll-based event sources also depends on additional factors, as noted following:

- **Poll-based event sources that are stream-based**
 - Amazon Kinesis Data Streams
 - Amazon DynamoDB

For Lambda functions that process Kinesis or DynamoDB streams the number of shards is the unit of concurrency. If your stream has 100 active shards, there will be at most 100 Lambda function invocations running concurrently. This is because Lambda processes each shard's events in sequence.

- **Poll-based event sources that are not stream-based:** For Lambda functions that process Amazon SQS queues, AWS Lambda will automatically scale the polling on the queue until the maximum concurrency level is reached, where each message batch can be considered a single concurrent unit. AWS Lambda's automatic scaling behavior is designed to keep polling costs low when a queue is empty while simultaneously enabling you to achieve high throughput when the queue is being used heavily.

When an Amazon SQS event source mapping is initially enabled, Lambda begins long-polling the Amazon SQS queue. Long polling helps reduce the cost of polling Amazon Simple Queue Service by reducing the number of empty responses, while providing optimal processing latency when messages arrive.

As the influx of messages to a queue increases, AWS Lambda automatically scales up polling activity until the number of concurrent function executions reaches 1000, the account concurrency limit, or the (optional) function concurrency limit, whichever is lower. Amazon Simple Queue Service supports an initial burst of 5 concurrent function invocations and increases concurrency by 60 concurrent invocations per minute.

Note

[Account-level limits](#) are impacted by other functions in the account, and per-function concurrency applies to all events sent to a function. For more information, see [Managing Concurrency \(p. 37\)](#).

Request Rate

Request rate refers to the rate at which your Lambda function is invoked. For all services except the poll-based services, the request rate is the rate at which the event sources generate the events. For poll-based services, AWS Lambda calculates the request rate as follows:

```
request rate = number of concurrent executions / function duration
```

For example, if there are five active shards on a stream (that is, you have five Lambda functions running in parallel) and your Lambda function takes about two seconds, the request rate is 2.5 requests/second.

Automatic Scaling

AWS Lambda dynamically scales function execution in response to increased traffic, up to your [concurrency limit \(p. 7\)](#). Under sustained load, your function's concurrency bursts to an initial level between 500 and 3000 concurrent executions that varies per region. After the initial burst, the function's

capacity increases by an additional 500 concurrent executions each minute until either the load is accommodated, or the total concurrency of all functions in the region hits the limit.

Region	Initial concurrency burst
US East (Ohio), US West (N. California), Canada (Central)	500
US West (Oregon), US East (N. Virginia)	3000
Asia Pacific (Seoul), Asia Pacific (Mumbai), Asia Pacific (Singapore), Asia Pacific (Sydney)	500
Asia Pacific (Tokyo)	1000
EU (London), EU (Paris), EU (Stockholm)	500
EU (Frankfurt)	1000
EU (Ireland)	3000
South America (São Paulo)	500
China (Beijing), China (Ningxia)	500
AWS GovCloud (US-West)	500

Note

If your function is connected to a VPC, the [Amazon VPC network interface limit](#) can prevent it from scaling. For more information, see [Configuring a Lambda Function to Access Resources in an Amazon VPC \(p. 68\)](#).

To limit scaling, you can configure functions with *reserved concurrency*. For more information, see [Managing Concurrency \(p. 37\)](#).

AWS Lambda Function Dead Letter Queues

Any Lambda function invoked **asynchronously** is retried twice before the event is discarded. If the retries fail and you're unsure why, use Dead Letter Queues (DLQ) to direct unprocessed events to an Amazon SQS queue or an Amazon SNS topic to analyze the failure.

AWS Lambda directs events that cannot be processed to the specified [Amazon SNS topic](#) or [Amazon SQS queue](#). Functions that don't specify a DLQ will discard events after they have exhausted their retries. For more information about retry policies, see [AWS Lambda Retry Behavior \(p. 85\)](#).

You configure a DLQ by specifying the Amazon Resource Name `TargetArn` value on the Lambda function's `DeadLetterConfig` parameter.

```
{
  "Code": {
    "ZipFile": blob,
    "S3Bucket": "string",
    "S3Key": "string",
    "S3ObjectVersion": "string"
  },
  "Description": "string",
  "FunctionName": "string",
  "Handler": "string",
  "MemorySize": 128,
  "Role": "arn:aws:iam::123456789012:role/LambdaRole"
}
```

```

    "Handler": "string",
    "MemorySize": number,
    "Role": "string",
    "Runtime": "string",
    "Timeout": number
    "Publish": bool,
    "DeadLetterConfig": {
        "TargetArn": "string"
    }
}

```

In addition, you need to add permissions to the [execution role \(p. 9\)](#) of your Lambda function, depending on which service you have directed unprocessed events:

- **For Amazon SQS:** [SendMessage](#)
- **For Amazon SNS:** [Publish](#)

The payload written to the DLQ target ARN is the original event payload with no modifications to the message body. The attributes of the message contain information to help you understand why the event wasn't processed:

DLQ Message Attributes

Name	Type	Value
RequestID	String	Unique request identifier
ErrorCode	Number	3-digit HTTP error code
ErrorMessage	String	Error message (truncated to 1 KB)

DLQ messages can fail to reach their target due to permissions issues, or if the total size of the message exceeds the limit for the target queue or topic. For example, if an Amazon SNS notification with a body close to 256 KB triggers a function that results in an error, the additional event data added by Amazon SNS, combined with the attributes added by Lambda, can cause the message to exceed the maximum size allowed in the DLQ. When it can't write to the DLQ, Lambda deletes the event and emits the [DeadLetterErrors \(p. 229\)](#) metric.



If you are using Amazon SQS as an event source, configure a DLQ on the Amazon SQS queue itself and not the Lambda function. For more information, see [Using AWS Lambda with Amazon SQS \(p. 216\)](#).

Using AWS Lambda with the AWS Command Line Interface

You can use the AWS Command Line Interface to manage functions and other AWS Lambda resources. The AWS CLI uses the AWS SDK for Python (Boto) to interact with the Lambda API. You can use it to

learn about the API, and apply that knowledge in building applications that use Lambda with the AWS SDK.

In this tutorial, you manage and invoke Lambda functions with the AWS CLI.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Getting Started with AWS Lambda \(p. 3\)](#) to create your first Lambda function.

To follow the procedures in this guide, you will need a command line terminal or shell to run commands. Commands are shown in listings preceded by a prompt symbol (\$) and the name of the current directory, when appropriate:

```
~/lambda-project$ this is a command  
this is output
```

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

On Linux and macOS, use your preferred shell and package manager. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the Execution Role

Create the [execution role \(p. 9\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – AWS Lambda.**
 - **Permissions – AWSLambdaBasicExecutionRole.**
 - **Role name – lambda-cli-role.**

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

Create the Function

The following example code receives an event as input and logs some of the incoming event data to CloudWatch Logs.

Example index.js

```
console.log('Loading function');

exports.handler = function(event, context, callback) {
    console.log('value1 =', event.key1);
    console.log('value2 =', event.key2);
    console.log('value3 =', event.key3);
    callback(null, "Success");
};
```

To create the function

1. Copy the sample code into a file named `index.js`.
2. Create a deployment package.

```
$ zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command.

```
$ aws lambda create-function --function-name helloworld \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs8.10 \
--role arn:aws:iam::123456789012:role/lambda-cli-role
{
    "FunctionName": "helloworld",
    "CodeSize": 351,
    "MemorySize": 128,
    "FunctionArn": "function-arn",
    "Handler": "index.handler",
    "Role": "arn:aws:iam::account-id:role/LambdaExecRole",
    "Timeout": 3,
    "LastModified": "2015-04-07T22:02:58.854+0000",
    "Runtime": "nodejs8.10",
    "Description": ""
}
```

Invoke your Lambda function using the `invoke` command.

```
$ aws lambda invoke --function-name helloworld --log-type Tail \
--payload '{"key1":"value1", "key2":"value2", "key3":"value3"}' \
outputfile.txt
{
    "LogResult": "base64-encoded-log",
    "StatusCode": 200
}
```

By specifying the `--log-type` parameter, the command also requests the tail end of the log produced by the function. The log data in the response is base64-encoded. Use the `base64` program to decode the log.

```
$ echo base64-encoded-log | base64 --decode
START RequestId: 16d25499-d89f-11e4-9e64-5d70fce44801
2015-04-01T18:44:12.323Z 16d25499-d89f-11e4-9e64-5d70fce44801 value1 = value1
2015-04-01T18:44:12.323Z 16d25499-d89f-11e4-9e64-5d70fce44801 value2 = value2
2015-04-01T18:44:12.323Z 16d25499-d89f-11e4-9e64-5d70fce44801 value3 = value3
2015-04-01T18:44:12.323Z 16d25499-d89f-11e4-9e64-5d70fce44801 result: "value1"
END RequestId: 16d25499-d89f-11e4-9e64-5d70fce44801
REPORT RequestId: 16d25499-d89f-11e4-9e64-5d70fce44801
Duration: 13.35 ms      Billed Duration: 100 ms   Memory Size: 128 MB
Max Memory Used: 9 MB
```

Because you invoked the function using the default invocation type (`RequestResponse`), the connection stays open until execution completes. Lambda writes the response to the output file.

List the Lambda Functions in Your Account

Execute the following AWS CLI `list-functions` command to retrieve a list of functions that you have created.

```
$ aws lambda list-functions --max-items 10
{
    "Functions": [
        {
            "FunctionName": "helloworld",
            "MemorySize": 128,
            "CodeSize": 412,
            "FunctionArn": "arn:aws:lambda:us-east-1:account-id:function:ProcessKinesisRecords",
            "Handler": "ProcessKinesisRecords.handler",
            "Role": "arn:aws:iam::account-id:role/LambdaExecRole",
            "Timeout": 3,
            "LastModified": "2015-02-22T21:03:01.172+0000",
            "Runtime": "nodejs6.10",
            "Description": ""
        },
        {
            "FunctionName": "ProcessKinesisRecords",
            "MemorySize": 128,
            "CodeSize": 412,
            "FunctionArn": "arn:aws:lambda:us-east-1:account-id:function:ProcessKinesisRecords",
            "Handler": "ProcessKinesisRecords.handler",
            "Role": "arn:aws:iam::account-id:role/lambda-execute-test-kinesis",
            "Timeout": 3,
            "LastModified": "2015-02-22T21:03:01.172+0000",
            "Runtime": "nodejs6.10",
            "Description": ""
        },
        ...
    ],
    "NextMarker": null
}
```

In response, Lambda returns a list of up to 10 functions. If there are more functions you can retrieve, `NextMarker` provides a marker you can use in the next `list-functions` request; otherwise, the value is null. The following `list-functions` AWS CLI command is an example that shows the `--marker` parameter.

```
$ aws lambda list-functions --max-items 10 \
--marker value-of-NextMarker-from-previous-response
```

Retrieve a Lambda Function

The Lambda CLI `get-function` command returns Lambda function metadata and a presigned URL that you can use to download the function's deployment package.

```
$ aws lambda get-function --function-name helloworld
{
    "Code": {
        "RepositoryType": "S3",
        "Location": "pre-signed-url"
    },
    "Configuration": {
        "FunctionName": "helloworld",
        "MemorySize": 128,
        "CodeSize": 287,
        "FunctionArn": "arn:aws:lambda:us-west-2:account-id:function:helloworld",
        "Handler": "index.handler",
        "Role": "arn:aws:iam::account-id:role/LambdaExecRole",
        "Timeout": 3,
```

```
        "LastModified": "2015-04-07T22:02:58.854+0000",
        "Runtime": "nodejs8.10",
        "Description": ""
    }
}
```

For more information, see [GetFunction \(p. 382\)](#).

Clean Up

Execute the following `delete-function` command to delete the `helloworld` function.

```
$ aws lambda delete-function --function-name helloworld
```

Delete the IAM role you created in the IAM console. For information about deleting a role, see [Deleting Roles or Instance Profiles](#) in the *IAM User Guide*.

Invoking Lambda Functions with the AWS Mobile SDK for Android

You can call a Lambda function from a mobile application. Put business logic in functions to separate its development lifecycle from that of front-end clients, making mobile applications less complex to develop and maintain. With the Mobile SDK for Android, you [use Amazon Cognito to authenticate users and authorize requests \(p. 93\)](#).

When you invoke a function from a mobile application, you choose the event structure, [invocation type \(p. 82\)](#), and permission model. You can use [aliases \(p. 51\)](#) to enable seamless updates to your function code, but otherwise the function and application are tightly coupled. As you add more functions, you can create an API layer to decouple your function code from your front-end clients and improve performance.

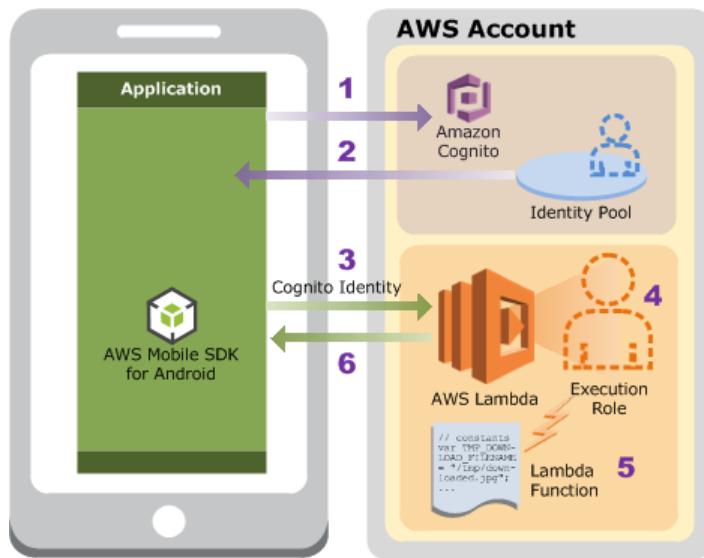
To create a fully-featured web API for your mobile and web applications, use Amazon API Gateway. With API Gateway, you can add custom authorizers, throttle requests, and cache results for all of your functions. For more information, see [Using AWS Lambda with Amazon API Gateway \(p. 135\)](#).

Topics

- [Tutorial: Using AWS Lambda with the Mobile SDK for Android \(p. 93\)](#)
- [Sample Function Code \(p. 99\)](#)

Tutorial: Using AWS Lambda with the Mobile SDK for Android

In this tutorial, you create a simple Android mobile application that uses Amazon Cognito to get credentials and invokes a Lambda function.



The mobile application retrieves AWS credentials from an Amazon Cognito identity pool and uses them to invoke a Lambda function with an event that contains request data. The function processes the request and returns a response to the front-end.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Getting Started with AWS Lambda \(p. 3\)](#) to create your first Lambda function.

To follow the procedures in this guide, you will need a command line terminal or shell to run commands. Commands are shown in listings preceded by a prompt symbol (\$) and the name of the current directory, when appropriate:

```
~/lambda-project$ this is a command
this is output
```

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

On Linux and macOS, use your preferred shell and package manager. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the Execution Role

Create the [execution role \(p. 9\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – AWS Lambda.**
 - **Permissions – AWSLambdaBasicExecutionRole.**
 - **Role name – lambda-android-role.**

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

Create the Function

The following example uses data to generate a string response.

Note

For sample code in other languages, see [Sample Function Code \(p. 99\)](#).

Example index.js

```
exports.handler = function(event, context, callback) {
    console.log("Received event: ", event);
    var data = {
        "greetings": "Hello, " + event.firstName + " " + event.lastName + "."
    };
    callback(null, data);
}
```

To create the function

1. Copy the sample code into a file named `index.js`.
2. Create a deployment package.

```
$ zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command.

```
$ aws lambda create-function --function-name AndroidBackendLambdaFunction \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs8.10 \
--role arn:aws:iam::123456789012:role/lambda-android-role
```

Test the Lambda Function

Invoke the function manually using the sample event data.

To test the Lambda function (AWS CLI)

1. Save the following sample event JSON in a file, `input.txt`.

```
{ "firstName": "first-name", "lastName": "last-name" }
```

2. Execute the following `invoke` command:

```
$ aws lambda invoke --function-name AndroidBackendLambdaFunction \
--payload file://file-path/input.txt outputfile.txt
```

Create an Amazon Cognito Identity Pool

In this section, you create an Amazon Cognito identity pool. The identity pool has two IAM roles. You update the IAM role for unauthenticated users and grant permissions to execute the `AndroidBackendLambdaFunction` Lambda function.

For more information about IAM roles, see [IAM Roles](#) in the *IAM User Guide*. For more information about Amazon Cognito services, see the [Amazon Cognito](#) product detail page.

To create an identity pool

1. Open the [Amazon Cognito console](#).
2. Create a new identity pool called `JavaFunctionAndroidEventHandlerPool`. Before you follow the procedure to create an identity pool, note the following:
 - The identity pool you are creating must allow access to unauthenticated identities because our example mobile application does not require a user log in. Therefore, make sure to select the **Enable access to unauthenticated identities** option.
 - Add the following statement to the permission policy associated with the unauthenticated identities.

```
{  
    "Effect": "Allow",  
    "Action": [  
        "lambda:InvokeFunction"  
    ],  
    "Resource": [  
        "arn:aws:lambda:us-  
east-1:123456789012:function:AndroidBackendLambdaFunction"  
    ]  
}
```

The resulting policy will be as follows:

```
{  
    "Version":"2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "mobileanalytics:PutEvents",  
                "cognito-sync:*"  
            ],  
            "Resource": [  
                "*"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "lambda:invokefunction"  
            ],  
            "Resource": [  
                "arn:aws:lambda:us-east-1:account-  
id:function:AndroidBackendLambdaFunction"  
            ]  
        }  
    ]  
}
```

For instructions about how to create an identity pool, log in to the [Amazon Cognito console](#) and follow the **New Identity Pool** wizard.

3. Note the identity pool ID. You specify this ID in your mobile application you create in the next section. The app uses this ID when it sends request to Amazon Cognito to request for temporary security credentials.

Create an Android Application

Create a simple Android mobile application that generates events and invokes Lambda functions by passing the event data as parameters.

The following instructions have been verified using Android studio.

1. Create a new Android project called `AndroidEventGenerator` using the following configuration:

- Select the **Phone and Tablet** platform.
- Choose **Blank Activity**.

2. In the `build.gradle (Module:app)` file, add the following in the dependencies section:

```
compile 'com.amazonaws:aws-android-sdk-core:2.2.+'  
compile 'com.amazonaws:aws-android-sdk-lambda:2.2.+'
```

3. Build the project so that the required dependencies are downloaded, as needed.
4. In the Android application manifest (`AndroidManifest.xml`), add the following permissions so that your application can connect to the Internet. You can add them just before the `</manifest>` end tag.

```
<uses-permission android:name="android.permission.INTERNET" />  
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

5. In `MainActivity`, add the following imports:

```
import com.amazonaws.mobileconnectors.lambdainvoker.*;  
import com.amazonaws.auth.CognitoCachingCredentialsProvider;  
import com.amazonaws.regions.Regions;
```

6. In the package section, add the following two classes (`RequestClass` and `ResponseClass`). Note that the POJO is same as the POJO you created in your Lambda function in the preceding section.

- `RequestClass`. The instances of this class act as the POJO (Plain Old Java Object) for event data which consists of first and last name. If you are using Java example for your Lambda function you created in the preceding section, this POJO is same as the POJO you created in your Lambda function code.

```
package com.example....lambdaeventgenerator;  
public class RequestClass {  
    String firstName;  
    String lastName;  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
  
    public String getLastName() {  
        return lastName;  
    }  
  
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }  
  
    public RequestClass(String firstName, String lastName) {
```

```
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public RequestClass() {
    }
}
```

- ResponseClass

```
package com.example....lambdaeventgenerator;
public class ResponseClass {
    String greetings;

    public String getGreetings() {
        return greetings;
    }

    public void setGreetings(String greetings) {
        this.greetings = greetings;
    }

    public ResponseClass(String greetings) {
        this.greetings = greetings;
    }

    public ResponseClass() {
    }
}
```

7. In the same package, create interface called MyInterface for invoking the AndroidBackendLambdaFunction Lambda function.

```
package com.example....lambdaeventgenerator;
import com.amazonaws.mobileconnectors.lambdainvoker.LambdaFunction;
public interface MyInterface {

    /**
     * Invoke the Lambda function "AndroidBackendLambdaFunction".
     * The function name is the method name.
     */
    @LambdaFunction
    ResponseClass AndroidBackendLambdaFunction(RequestClass request);

}
```

The `@LambdaFunction` annotation in the code maps the specific client method to the same-name Lambda function. For more information about this annotation, see [AWS Lambda](#) in the [AWS Mobile SDK for Android Developer Guide](#).

8. To keep the application simple, we are going to add code to invoke the Lambda function in the `onCreate()` event handler. In `MainActivity`, add the following code toward the end of the `onCreate()` code.

```
// Create an instance of CognitoCachingCredentialsProvider
CognitoCachingCredentialsProvider cognitoProvider = new
    CognitoCachingCredentialsProvider(
        this.getApplicationContext(), "identity-pool-id", Regions.US_WEST_2);

// Create LambdaInvokerFactory, to be used to instantiate the Lambda proxy.
LambdaInvokerFactory factory = new LambdaInvokerFactory(this.getApplicationContext(),
    Regions.US_WEST_2, cognitoProvider);
```

```
// Create the Lambda proxy object with a default Json data binder.  
// You can provide your own data binder by implementing  
// LambdaDataBinder.  
final MyInterface myInterface = factory.build(MyInterface.class);  
  
RequestClass request = new RequestClass("John", "Doe");  
// The Lambda function invocation results in a network call.  
// Make sure it is not called from the main thread.  
new AsyncTask<RequestClass, Void, ResponseClass>() {  
    @Override  
    protected ResponseClass doInBackground(RequestClass... params) {  
        // invoke "echo" method. In case it fails, it will throw a  
        // LambdaFunctionException.  
        try {  
            return myInterface.AndroidBackendLambdaFunction(params[0]);  
        } catch (LambdaFunctionException lfe) {  
            Log.e("Tag", "Failed to invoke echo", lfe);  
            return null;  
        }  
    }  
  
    @Override  
    protected void onPostExecute(ResponseClass result) {  
        if (result == null) {  
            return;  
        }  
  
        // Do a toast  
        Toast.makeText(MainActivity.this, result.getGreetings(),  
        Toast.LENGTH_LONG).show();  
    }  
}.execute(request);
```

9. Run the code and verify it as follows:

- The `Toast.makeText()` displays the response returned.
- Verify that CloudWatch Logs shows the log created by the Lambda function. It should show the event data (first name and last name). You can also verify this in the AWS Lambda console.

Sample Function Code

Sample code is available for the following languages.

Topics

- [Node.js \(p. 99\)](#)
- [Java \(p. 100\)](#)

Node.js

The following example uses data to generate a string response.

Example index.js

```
exports.handler = function(event, context, callback) {  
    console.log("Received event: ", event);  
    var data = {  
        "greetings": "Hello, " + event.firstName + " " + event.lastName + "."  
    };  
    callback(null, data);
```

```
}
```

Zip up the sample code to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Node.js \(p. 241\)](#).

Java

The following example uses data to generate a string response.

In the code, the handler (`myHandler`) uses the `RequestClass` and `ResponseClass` types for the input and output. The code provides implementation for these types.

Example HelloPojo.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;

public class HelloPojo {

    // Define two classes/POJOs for use with Lambda function.
    public static class RequestClass {
        String firstName;
        String lastName;

        public String getFirstName() {
            return firstName;
        }

        public void setFirstName(String firstName) {
            this.firstName = firstName;
        }

        public String getLastName() {
            return lastName;
        }

        public void setLastName(String lastName) {
            this.lastName = lastName;
        }

        public RequestClass(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }

        public RequestClass() {
        }
    }

    public static class ResponseClass {
        String greetings;

        public String getGreetings() {
            return greetings;
        }

        public void setGreetings(String greetings) {
            this.greetings = greetings;
        }

        public ResponseClass(String greetings) {
            this.greetings = greetings;
        }
    }
}
```

```
        }

    public ResponseClass() {
        }

    }

    public static ResponseClass myHandler(RequestClass request, Context context){
        String greetingString = String.format("Hello %s, %s.", request.firstName,
request.lastName);
        context.getLogger().log(greetingString);
        return new ResponseClass(greetingString);
    }
}
```

Dependencies

- `aws-lambda-java-core`

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Java \(p. 264\)](#).

AWS Lambda Runtimes

AWS Lambda supports multiple languages through the use of runtimes. You choose a runtime when you create a function, and you can change runtimes by updating your function's configuration. The underlying execution environment provides additional libraries and [environment variables \(p. 103\)](#) that you can access from your function code.

Amazon Linux

- AMI – [amzn-ami-hvm-2017.03.1.20170812-x86_64-gp2](#)
- Linux kernel – 4.14.106-92.87.amzn2.x86_64

Note

Lambda is upgrading to Amazon Linux 2018.03. See [Upcoming updates to the AWS Lambda and AWS Lambda@Edge execution environment](#) for details.

Amazon Linux 2

- AMI – [amzn2-ami-hvm-2.0.20190313-x86_64-gp2](#)
- Linux kernel – 4.14.106-92.87.amzn2.x86_64

When your function is invoked, Lambda attempts to re-use the execution environment from a previous invocation if one is available. This saves time preparing the execution environment, and allows you to save resources like database connections and temporary files in the [execution context \(p. 104\)](#) to avoid creating them every time your function runs.

A runtime can support a single version of a language, multiple versions of a language, or multiple languages. Runtimes specific to a language or framework version are [deprecated \(p. 105\)](#) when the version reaches end of life.

Node.js Runtimes

Name	Identifier	Node.js Version	AWS SDK for JavaScript	Operating System
Node.js 10	nodejs10.x	10.15	2.437.0	Amazon Linux 2
Node.js 8.10	nodejs8.10	8.10	2.290.0	Amazon Linux

Python Runtimes

Name	Identifier	AWS SDK for Python	Operating System
Python 3.6	python3.6	boto3-1.7.74 botocore-1.10.74	Amazon Linux
Python 3.7	python3.7	boto3-1.9.42 botocore-1.12.42	Amazon Linux
Python 2.7	python2.7	N/A	Amazon Linux

Ruby Runtimes

Name	Identifier	Operating System
Ruby 2.5	ruby2.5	Amazon Linux

Java Runtimes

Name	Identifier	JDK	Operating System
Java 8	java8	java-1.8.0-openjdk	Amazon Linux

Go Runtimes

Name	Identifier	Operating System
Go 1.x	go1.x	Amazon Linux

.NET Runtimes

Name	Identifier	Languages	Operating System
.NET Core 2.1	dotnetcore2.1	C# PowerShell Core 6.0	Amazon Linux
.NET Core 1.0	dotnetcore1.0	C#	Amazon Linux

To use other languages in Lambda, you can implement a [custom runtime \(p. 106\)](#). The Lambda execution environment provides a [runtime interface \(p. 108\)](#) for getting invocation events and sending responses. You can deploy a custom runtime alongside your function code, or in a [layer \(p. 64\)](#).

Topics

- [Environment Variables Available to Lambda Functions \(p. 103\)](#)
- [AWS Lambda Execution Context \(p. 104\)](#)
- [Runtime Support Policy \(p. 105\)](#)
- [Custom AWS Lambda Runtimes \(p. 106\)](#)
- [AWS Lambda Runtime Interface \(p. 108\)](#)
- [Tutorial – Publishing a Custom Runtime \(p. 110\)](#)

Environment Variables Available to Lambda Functions

The following is a list of environment variables that are part of the AWS Lambda execution environment and made available to Lambda functions. The table below indicates which ones are reserved by AWS Lambda and can't be changed, as well as which ones you can set when creating your Lambda function. For more information on using environment variables with your Lambda function, see [AWS Lambda Environment Variables \(p. 40\)](#).

Lambda Environment Variables

Key	Reserved	Value
_HANDLER	Yes	The handler location configured on the function.
AWS_REGION	Yes	The AWS region where the Lambda function is executed.
AWS_EXECUTION_ENV	Yes	The runtime identifier (p. 102) , prefixed by AWS_Lambda_. For example, AWS_Lambda_java8.
AWS_LAMBDA_FUNCTION_NAME	Yes	The name of the function.
AWS_LAMBDA_FUNCTION_MEMORY_SIZE	Yes	The amount of memory available to the function in MB.
AWS_LAMBDA_FUNCTION_VERSION	Yes	The version of the function being executed.
AWS_LAMBDA_LOG_GROUP_NAME	Yes	The name of the Amazon CloudWatch Logs group and stream for the function.
AWS_LAMBDA_LOG_STREAM_NAME		
AWS_ACCESS_KEY_ID	Yes	Access keys obtained from the function's execution role (p. 9) .
AWS_SECRET_ACCESS_KEY		
AWS_SESSION_TOKEN		
LANG	No	en_US.UTF-8. This is the locale of the runtime.
TZ	Yes	The environment's timezone (UTC). The execution environment uses NTP to synchronize the system clock.
LAMBDA_TASK_ROOT	Yes	The path to your Lambda function code.
LAMBDA_RUNTIME_DIR	Yes	The path to runtime libraries.
PATH	No	/usr/local/bin:/usr/bin/:/bin:/opt/bin
LD_LIBRARY_PATH	No	/lib64:/usr/lib64:\$LAMBDA_RUNTIME_DIR:\$LAMBDA_RUNTIME_DIR/lib:\$LAMBDA_TASK_ROOT:\$LAMBDA_TASK_ROOT/lib:/opt/lib
NODE_PATH	No	(Node.js) /opt/nodejs/node8/node_modules/:/opt/nodejs/node_modules:\$LAMBDA_RUNTIME_DIR/node_modules
PYTHONPATH	No	(Python) \$LAMBDA_RUNTIME_DIR.
GEM_PATH	No	(Ruby) \$LAMBDA_TASK_ROOT/vendor/bundle/ruby/2.5.0:/opt/ruby/gems/2.5.0.
AWS_LAMBDA_RUNTIME_API	Yes	(custom runtime) The host and port of the runtime API (p. 108) .

AWS Lambda Execution Context

When AWS Lambda executes your Lambda function, it provisions and manages the resources needed to run your Lambda function. When you create a Lambda function, you specify configuration information, such as the amount of memory and maximum execution time that you want to allow for your Lambda function. When a Lambda function is invoked, AWS Lambda launches an execution context based on

the configuration settings you provide. The execution context is a temporary runtime environment that initializes any external dependencies of your Lambda function code, such as database connections or HTTP endpoints. This affords subsequent invocations better performance because there is no need to "cold-start" or initialize those external dependencies, as explained below.

It takes time to set up an execution context and do the necessary "bootstrapping", which adds some latency each time the Lambda function is invoked. You typically see this latency when a Lambda function is invoked for the first time or after it has been updated because AWS Lambda tries to reuse the execution context for subsequent invocations of the Lambda function.

After a Lambda function is executed, AWS Lambda maintains the execution context for some time in anticipation of another Lambda function invocation. In effect, the service freezes the execution context after a Lambda function completes, and thaws the context for reuse, if AWS Lambda chooses to reuse the context when the Lambda function is invoked again. This execution context reuse approach has the following implications:

- Any declarations in your Lambda function code (outside the handler code, see [Programming Model \(p. 31\)](#)) remains initialized, providing additional optimization when the function is invoked again. For example, if your Lambda function establishes a database connection, instead of reestablishing the connection, the original connection is used in subsequent invocations. We suggest adding logic in your code to check if a connection exists before creating one.
- Each execution context provides 512 MB of additional disk space in the /tmp directory. The directory content remains when the execution context is frozen, providing transient cache that can be used for multiple invocations. You can add extra code to check if the cache has the data that you stored. For information on deployment limits, see [AWS Lambda Limits \(p. 7\)](#).
- Background processes or callbacks initiated by your Lambda function that did not complete when the function ended resume if AWS Lambda chooses to reuse the execution context. You should make sure any background processes or callbacks (in case of Node.js) in your code are complete before the code exits.

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

Runtime Support Policy

AWS Lambda runtimes are built around a combination of operating system, programming language, and software libraries that are subject to maintenance and security updates. When a component of a runtime is no longer supported for security updates, Lambda deprecates the runtime.

Deprecation occurs in two phases. During the first phase, you can no longer create functions that use the deprecated runtime. For at least 30 days, you can continue to update existing functions that use the deprecated runtime. After this period, both function creation and updates are disabled permanently. However, the function continues to be available to process invocation events.

Deprecation Schedule

Name	Identifier	End of Life	Deprecation (Create)	Deprecation (Update)
Node.js 0.10	nodejs	October 31, 2016	October 31, 2016	October 31, 2016
Node.js 4.3	nodejs4.3 nodejs4.3-edge	April 30, 2018	December 15, 2018	April 30, 2019

Name	Identifier	End of Life	Deprecation (Create)	Deprecation (Update)
Node.js 6.10	nodejs6.10	April 30, 2019	April 30, 2019	June 30, 2019
.NET Core 2.0	dotnetcore2.0	April 30, 2019	April 30, 2019	May 30, 2019

In most cases, the end-of-life date of a language version or operating system is known well in advance. If you have functions running on a runtime that will be deprecated in the next 60 days, Lambda notifies you by email that you should prepare by migrating your function to a supported runtime. In some cases, such as security issues that require a backwards-incompatible update, or software that doesn't support a long-term support (LTS) schedule, advance notice might not be possible.

After a runtime is deprecated, Lambda might retire it completely at any time by disabling invocation. Deprecated runtimes aren't eligible for security updates or technical support. Before retiring a runtime, Lambda sends additional notifications to affected customers. No runtimes are scheduled to be retired at this time.

Custom AWS Lambda Runtimes

You can implement an AWS Lambda runtime in any programming language. A runtime is a program that runs a Lambda function's handler method when the function is invoked. You can include a runtime in your function's deployment package in the form of an executable file named `bootstrap`.

A runtime is responsible for running the function's setup code, reading the handler name from an environment variable, and reading invocation events from the Lambda runtime API. The runtime passes the event data to the function handler, and posts the response from the handler back to Lambda.

Your custom runtime runs in the standard Lambda [execution environment \(p. 102\)](#). It can be a shell script, a script in a language that's included in Amazon Linux, or a binary executable file that's compiled in Amazon Linux.

To get started with custom runtimes, see [Tutorial – Publishing a Custom Runtime \(p. 110\)](#). You can also explore a custom runtime implemented in C++ at [awslabs/aws-lambda-cpp](#) on GitHub.

Topics

- [Using a Custom Runtime \(p. 106\)](#)
- [Building a Custom Runtime \(p. 107\)](#)

Using a Custom Runtime

To use a custom runtime, set your function's runtime to `provided`. The runtime can be included in your function's deployment package, or in a [layer \(p. 64\)](#).

Example `function.zip`

```
.
### bootstrap
### function.sh
```

If there's a file named `bootstrap` in your deployment package, Lambda executes that file. If not, Lambda looks for a runtime in the function's layers. If the `bootstrap` file isn't found or isn't executable, your function returns an error upon invocation.

Building a Custom Runtime

A custom runtime's entry point is an executable file named `bootstrap`. The bootstrap file can be the runtime, or it can invoke another file that creates the runtime. The following example uses a bundled version of Node.js to execute a JavaScript runtime in a separate file named `runtime.js`.

Example `bootstrap`

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

Your runtime code is responsible for completing some initialization tasks. Then it processes invocation events in a loop until it's terminated. The initialization tasks run once [per instance of the function \(p. 104\)](#) to prepare the environment to handle invocations.

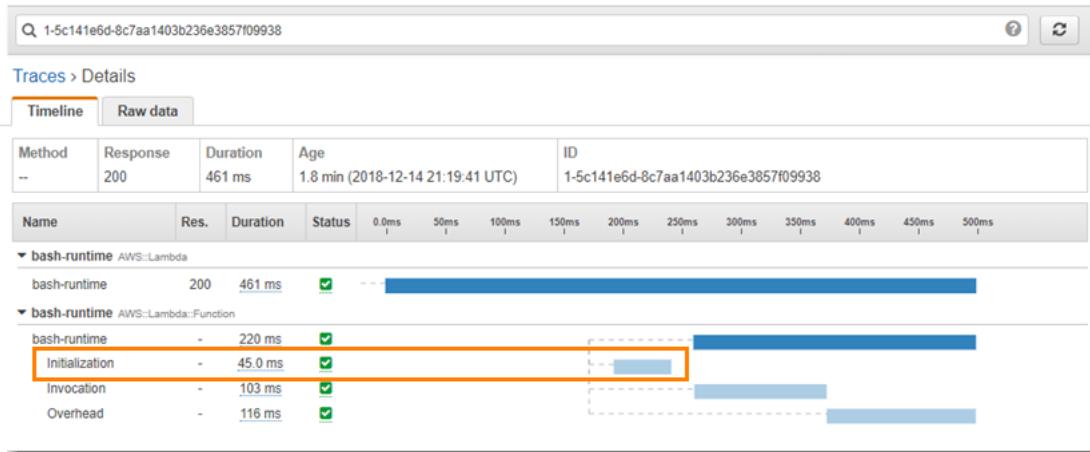
Initialization Tasks

- **Retrieve settings** – Read environment variables to get details about the function and environment.
 - `_HANDLER` – The location to the handler, from the function's configuration. The standard format is `file.method`, where `file` is the name of the file without an extension, and `method` is the name of a method or function that's defined in the file.
 - `LAMBDA_TASK_ROOT` – The directory that contains the function code.
 - `AWS_LAMBDA_RUNTIME_API` – The host and port of the runtime API.
- See [Environment Variables Available to Lambda Functions \(p. 103\)](#) for a full list of available variables.
- **Initialize the function** – Load the handler file and run any global or static code that it contains. Functions should create static resources like SDK clients and database connections once, and reuse them for multiple invocations.
- **Handle errors** – If an error occurs, call the [initialization error \(p. 110\)](#) API and exit immediately.

Initialization counts towards billed execution time and timeout. When an execution triggers the initialization of a new instance of your function, you can see the initialization time in the logs and [AWS X-Ray trace \(p. 232\)](#).

Example Log

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



While it runs, a runtime uses the [Lambda runtime interface \(p. 108\)](#) to manage incoming events and report errors. After completing initialization tasks, the runtime processes incoming events in a loop.

Processing Tasks

- **Get an event** – Call the [next invocation \(p. 109\)](#) API to get the next event. The response body contains the event data. Response headers contain the request ID and other information.
- **Propagate the tracing header** – Get the X-Ray tracing header from the `Lambda-Runtime-Trace-Id` header in the API response. Set the `_X_AMZN_TRACE_ID` environment variable with the same value for the X-Ray SDK to use.
- **Create a context object** – Create an object with context information from environment variables and headers in the API response.
- **Invoke the function handler** – Pass the event and context object to the handler.
- **Handle the response** – Call the [invocation response \(p. 109\)](#) API to post the response from the handler.
- **Handle errors** – If an error occurs, call the [invocation error \(p. 110\)](#) API.
- **Cleanup** – Release unused resources, send data to other services, or perform additional tasks before getting the next event.

You can include the runtime in your function's deployment package, or distribute the runtime separately in a function layer. For an example walkthrough, see [???](#) (p. 110).

AWS Lambda Runtime Interface

AWS Lambda provides an HTTP API for [custom runtimes \(p. 106\)](#) to receive invocation events from Lambda and send response data back within the Lambda [execution environment \(p. 102\)](#).

The OpenAPI specification for the runtime API version **2018-06-01** is available here: [runtime-api.zip](#)

Runtimes get an endpoint from the `AWS_LAMBDA_RUNTIME_API` environment variable, add the API version, and use the following resource paths to interact with the API.

Example Request

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

Resources

- [Next Invocation \(p. 109\)](#)
- [Invocation Response \(p. 109\)](#)
- [Invocation Error \(p. 110\)](#)
- [Initialization Error \(p. 110\)](#)

Next Invocation

Path – /runtime/invocation/next

Method – GET

Retrieves an invocation event. The response body contains the payload from the invocation, which is a JSON document that contains event data from the function trigger. The response headers contain additional data about the invocation.

Response Headers

- **Lambda-Runtime-Aws-Request-Id** – The request ID, which identifies the request that triggered the function invocation.

For example, 8476a536-e9f4-11e8-9739-2dfe598c3fcfd.
- **Lambda-Runtime-Deadline-Ms** – The date that the function times out in Unix time milliseconds.

For example, 1542409706888.
- **Lambda-Runtime-Invoked-Function-Arn** – The ARN of the Lambda function, version, or alias that's specified in the invocation.

For example, arn:aws:lambda:us-east-2:123456789012:function:custom-runtime.
- **Lambda-Runtime-Trace-Id** – The [AWS X-Ray tracing header](#).

For example, Root=1-5bef4de7-ad49b0e87f6ef6c87fc2e700;Parent=9a9197af755a6419;Sampled=1.
- **Lambda-Runtime-Client-Context** – For invocations from the AWS Mobile SDK, data about the client application and device.
- **Lambda-Runtime-Cognito-Identity** – For invocations from the AWS Mobile SDK, data about the Amazon Cognito identity provider.

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

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

Invocation Response

Path – /runtime/invocation/*AwsRequestId*/response

Method – POST

Sends an invocation response to Lambda. After the runtime invokes the function handler, it posts the response from the function to the invocation response path. For synchronous invocations, Lambda then sends the response back to the client.

Example Success Request

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

Invocation Error

Path – /runtime/invocation/*AwsRequestId*/error

Method – POST

If the function returns an error, the runtime formats the error into a JSON document, and posts it to the invocation error path.

Example Request Body

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

Example Error Request

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

Initialization Error

Path – /runtime/init/error

Method – POST

If the runtime encounters an error during initialization, it posts an error message to the initialization error path.

Example Initialization Error Request

```
ERROR={"\\"errorMessage\"\": \\"Failed to load function.\\"", \\"errorType\"\":
\\"InvalidFunctionException\""}
curl -X POST "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/init/error" -d "$ERROR"
--header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

Tutorial – Publishing a Custom Runtime

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

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Getting Started with AWS Lambda \(p. 3\)](#) to create your first Lambda function.

To follow the procedures in this guide, you will need a command line terminal or shell to run commands. Commands are shown in listings preceded by a prompt symbol (\$) and the name of the current directory, when appropriate:

```
~/lambda-project$ this is a command  
this is output
```

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

On Linux and macOS, use your preferred shell and package manager. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

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

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – Lambda.**
 - **Permissions – AWSLambdaBasicExecutionRole.**
 - **Role name – lambda-role.**

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

Create a Function

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

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

Example bootstrap

```
#!/bin/sh  
  
set -euo pipefail  
  
# Initialization - load function handler  
source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"  
  
# Processing  
while true  
do
```

```
HEADERS="$(mktemp)"  
# Get an event  
EVENT_DATA=$(curl -ss -LD "$HEADERS" -X GET "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")  
REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d '[:space:]' | cut -d: -f2)  
  
# Execute the handler function from the script  
RESPONSE=$(($(_HANDLER" | cut -d. -f2) "$EVENT_DATA"))  
  
# Send the response  
curl -X POST "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/response" -d "$RESPONSE"  
done
```

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

Note

Runtimes have additional responsibilities, including error handling, and providing context information to the handler. For details, see [Building a Custom Runtime \(p. 107\)](#).

The script defines a handler function that takes event data, logs it to `stderr`, and returns it.

Example function.sh

```
function handler () {  
    EVENT_DATA=$1  
    echo "$EVENT_DATA" 1>&2;  
    RESPONSE="Echoing request: '$EVENT_DATA'"  
  
    echo $RESPONSE  
}
```

Save both files in a project directory named `runtime-tutorial`.

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

Make the files executable and add them to a ZIP archive.

```
runtime-tutorial$ chmod 755 function.sh bootstrap  
runtime-tutorial$ zip function.zip function.sh bootstrap  
adding: function.sh (deflated 24%)  
adding: bootstrap (deflated 39%)
```

Create a function named `bash-runtime`.

```
runtime-tutorial$ aws lambda create-function --function-name bash-runtime \  
--zip-file file:///function.zip --handler function.handler --runtime provided \  
--role arn:aws:iam::123456789012:role/lambda-role  
{  
    "FunctionName": "bash-runtime",  
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:bash-runtime",  
    "Runtime": "provided",  
    "Role": "arn:aws:iam::123456789012:role/lambda-role",  
    "Handler": "function.handler",  
    "CodeSize": 831,
```

```
"Description": "",  
"Timeout": 3,  
"MemorySize": 128,  
"LastModified": "2018-11-28T06:57:31.095+0000",  
"CodeSha256": "mv/xRv84LPCxdpcbKvmwuuFzwo7sLwUO1VxcUv3wKlM=",  
"Version": "$LATEST",  
"TracingConfig": {  
    "Mode": "PassThrough"  
},  
"RevisionId": "2e1d51b0-6144-4763-8e5c-7d5672a01713"  
}
```

Invoke the function and verify the response.

```
runtime-tutorial$ aws lambda invoke --function-name bash-runtime --payload  
'{"text":"Hello"}' response.txt  
{  
    "StatusCode": 200,  
    "ExecutedVersion": "$LATEST"  
}  
runtime-tutorial$ cat response.txt  
Echoing request: '{"text":"Hello"}'
```

Create a Layer

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

Create a layer archive that contains the bootstrap file.

```
runtime-tutorial$ zip runtime.zip bootstrap  
adding: bootstrap (deflated 39%)
```

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

```
runtime-tutorial$ aws lambda publish-layer-version --layer-name bash-runtime --zip-file  
fileb://runtime.zip  
{  
    "Content": {  
        "Location": "https://awslambda-us-west-2-layers.s3.us-west-2.amazonaws.com/  
snapshots/123456789012/bash-runtime-018c209b...",  
        "CodeSha256": "bXVLhHi+D3H1QbDARUVPrDwlC7bssPxySQqt1QZqusE=",  
        "CodeSize": 584,  
        "UncompressedCodeSize": 0  
    },  
    "LayerArn": "arn:aws:lambda:us-west-2:123456789012:layer:bash-runtime",  
    "LayerVersionArn": "arn:aws:lambda:us-west-2:123456789012:layer:bash-runtime:1",  
    "Description": "",  
    "CreatedDate": "2018-11-28T07:49:14.476+0000",  
    "Version": 1  
}
```

This creates the first version of the layer.

Update the Function

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

Update the function configuration to pull in the layer.

```
runtime-tutorial$ aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-west-2:123456789012:layer:bash-runtime:1
{
    "FunctionName": "bash-runtime",
    "Layers": [
        {
            "Arn": "arn:aws:lambda:us-west-2:123456789012:layer:bash-runtime:1",
            "CodeSize": 584,
            "UncompressedCodeSize": 679
        }
    ]
    ...
}
```

This adds the runtime to the function in the /opt directory. Lambda uses this runtime, but only if you remove it from the function's deployment package. Update the function code to only include the handler script.

```
runtime-tutorial$ zip function-only.zip function.sh
adding: function.sh (deflated 24%)
runtime-tutorial$ aws lambda update-function-code --function-name bash-runtime --zip-file
fileb://function-only.zip
{
    "FunctionName": "bash-runtime",
    "CodeSize": 270,
    "Layers": [
        {
            "Arn": "arn:aws:lambda:us-west-2:123456789012:layer:bash-runtime:7",
            "CodeSize": 584,
            "UncompressedCodeSize": 679
        }
    ]
    ...
}
```

Invoke the function to verify that it works with the runtime layer.

```
runtime-tutorial$ aws lambda invoke --function-name bash-runtime --payload
'{"text":"Hello"}' response.txt
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
runtime-tutorial$ cat response.txt
Echoing request: '{"text":"Hello"}'
```

Update the Runtime

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

Example bootstrap

```
#!/bin/sh

set -euo pipefail

echo "## Environment variables:"
```

```
env

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

Create a second version of the layer with the new code.

```
runtime-tutorial$ zip runtime.zip bootstrap
updating: bootstrap (deflated 39%)
runtime-tutorial$ aws lambda publish-layer-version --layer-name bash-runtime --zip-file
fileb://runtime.zip
```

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

```
runtime-tutorial$ aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-west-2:123456789012:layer:bash-runtime:2
```

Share the Layer

Add a permission statement to your runtime layer to share it with other accounts.

```
runtime-tutorial$ aws lambda add-layer-version-permission --layer-name bash-runtime --
version-number 2 \
--principal "*" --statement-id publish --action lambda:GetLayerVersion
{
    "Statement": "{\"Sid\":\"publish\",\"Effect\":\"Allow\",\"Principal\":\"*\",\"Action\":
\"lambda:GetLayerVersion\", \"Resource\":\"arn:aws:lambda:us-west-2:123456789012:layer:bash-
runtime:2\"}",
    "RevisionId": "9d5fe08e-2a1e-4981-b783-37ab551247ff"
}
```

You can add multiple statements that each grant permission to a single account, accounts in an organization, or all accounts.

Clean Up

Delete each version of the layer.

```
runtime-tutorial$ aws lambda delete-layer-version --layer-name bash-runtime --version-
number 1
runtime-tutorial$ aws lambda delete-layer-version --layer-name bash-runtime --version-
number 2
```

Because the function holds a reference to version 2 of the layer, it still exists in Lambda. The function continues to work, but functions can no longer be configured to use the deleted version. If you then modify the list of layers on the function, you must specify a new version or omit the deleted layer.

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

```
runtime-tutorial$ aws lambda delete-function --function-name bash-runtime
```

AWS Lambda Applications

An AWS Lambda application is a combination of Lambda functions, event sources, and other resources that work together to perform tasks. You can use AWS CloudFormation and other tools to collect your application's components into a single package that can be deployed and managed as one resource. Applications make your Lambda projects portable and enable you to integrate with additional developer tools, such as AWS CodePipeline, AWS CodeBuild, and the AWS Serverless Application Model command line interface (SAM CLI).

The [AWS Serverless Application Repository](#) provides a collection of Lambda applications that you can deploy in your account with a few clicks. The repository includes both ready-to-use applications and samples that you can use as a starting point for your own projects. You can also submit your own projects for inclusion.

[AWS CloudFormation](#) enables you to create a template that defines your application's resources and lets you manage the application as a *stack*. You can more safely add or modify resources in your application stack. If any part of an update fails, AWS CloudFormation automatically rolls back to the previous configuration. With AWS CloudFormation parameters, you can create multiple environments for your application from the same template.

The [AWS Serverless Application Model \(p. 119\)](#) (AWS SAM) is an extension for the AWS CloudFormation template language that lets you define serverless applications at a higher level. It abstracts away common tasks such as function role creation, which makes it easier to write templates. AWS SAM is supported directly by AWS CloudFormation, and includes additional functionality through the AWS CLI and AWS SAM CLI.

The [AWS CLI](#) and [AWS SAM CLI](#) are command line tools for managing Lambda application stacks. In addition to commands for managing application stacks with the AWS CloudFormation API, the AWS CLI supports higher-level commands that simplify tasks like uploading deployment packages and updating templates. The AWS SAM CLI provides additional functionality, including validating templates and testing locally.

Topics

- [Managing Applications in the AWS Lambda Console \(p. 116\)](#)
- [Using the AWS Serverless Application Model \(AWS SAM\) \(p. 119\)](#)
- [Error Processor Sample Application for AWS Lambda \(p. 119\)](#)
- [Building a Continuous Delivery Pipeline for a Lambda Application with AWS CodePipeline \(p. 123\)](#)
- [Best Practices for Working with AWS Lambda Functions \(p. 127\)](#)

Managing Applications in the AWS Lambda Console

The AWS Lambda console helps you monitor and manage your [Lambda applications \(p. 116\)](#). The **Applications** menu lists AWS CloudFormation stacks with Lambda functions. The menu includes stacks that you launch in AWS CloudFormation by using the AWS CloudFormation console, the AWS Serverless Application Repository, the AWS CLI, or the AWS SAM CLI.

To view a Lambda application

1. Open the [Lambda console](#).
2. Choose **Applications**.
3. Choose an application.

Name	Description	Last modified	Status
scorekeep-random-name	Create a Lambda function with permission to use SNS and X-Ray	20 days ago	UPDATE_COMPLETE
lambda-web-page	Set up API Gateway endpoint connected to Lambda function that returns HTML	23 days ago	CREATE_COMPLETE

The overview shows the following information about your application.

- **AWS CloudFormation template or SAM template** – The template that defines your application.
- **Resources** – The AWS resources that are defined in your application's template. To manage your application's Lambda functions, choose a function name from the list.

Monitoring Applications

The **Monitoring** tab shows an Amazon CloudWatch dashboard with aggregate metrics for the resources in your application.

To monitor a Lambda application

1. Open the [Lambda console](#).
2. Choose **Applications**.
3. Choose **Monitoring**.

By default, the Lambda console shows a basic dashboard. You can customize this page by defining custom dashboards in your application template. When your template includes one or more dashboards, the page shows your dashboards instead of the default dashboard. You can switch between dashboards with the drop-down menu on the top right of the page.

Custom Monitoring Dashboards

Customize your application monitoring page by adding one or more Amazon CloudWatch dashboards to your application template with the [AWS::CloudWatch::Dashboard](#) resource type. The following example creates a dashboard with a single widget that graphs the number of invocations of a function named `my-function`.

Example Function Dashboard Template

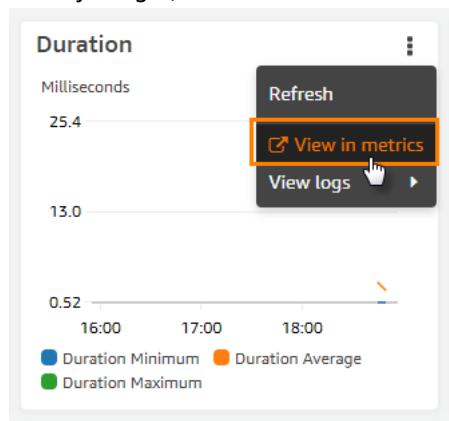
```
Resources:
  MyDashboard:
    Type: AWS::CloudWatch::Dashboard
    Properties:
      DashboardName: my-dashboard
```

```
DashboardBody: |
{
    "widgets": [
        {
            "type": "metric",
            "width": 12,
            "height": 6,
            "properties": {
                "metrics": [
                    [
                        {
                            "AWS/Lambda",
                            "Invocations",
                            "FunctionName",
                            "my-function",
                            {
                                "stat": "Sum",
                                "label": "MyFunction"
                            }
                        ],
                        [
                            {
                                "expression": "SUM(METRICS())",
                                "label": "Total Invocations"
                            }
                        ]
                    ],
                    "region": "us-east-1",
                    "title": "Invocations",
                    "view": "timeSeries",
                    "stacked": false
                }
            }
        ]
    ]
}
```

You can get the definition for any of the widgets in the default monitoring dashboard from the CloudWatch console.

To view a widget definition

1. Open the [Lambda console](#).
2. Choose **Applications**.
3. Choose an application that has the standard dashboard.
4. Choose **Monitoring**.
5. On any widget, choose **View in metrics** from the drop-down menu.



6. Choose Source.

For more information about authoring CloudWatch dashboards and widgets, see [Dashboard Body Structure and Syntax](#) in the *Amazon CloudWatch API Reference*.

Using the AWS Serverless Application Model (AWS SAM)

The AWS Serverless Application Model (AWS SAM) is an open-source framework you can use to build serverless applications on AWS. It consists of the AWS SAM template specification that you use to define your serverless applications, and the AWS SAM command line interface (AWS SAM CLI) that you use to build, test and deploy your serverless applications.

For more information about AWS SAM, see [What Is AWS SAM?](#).

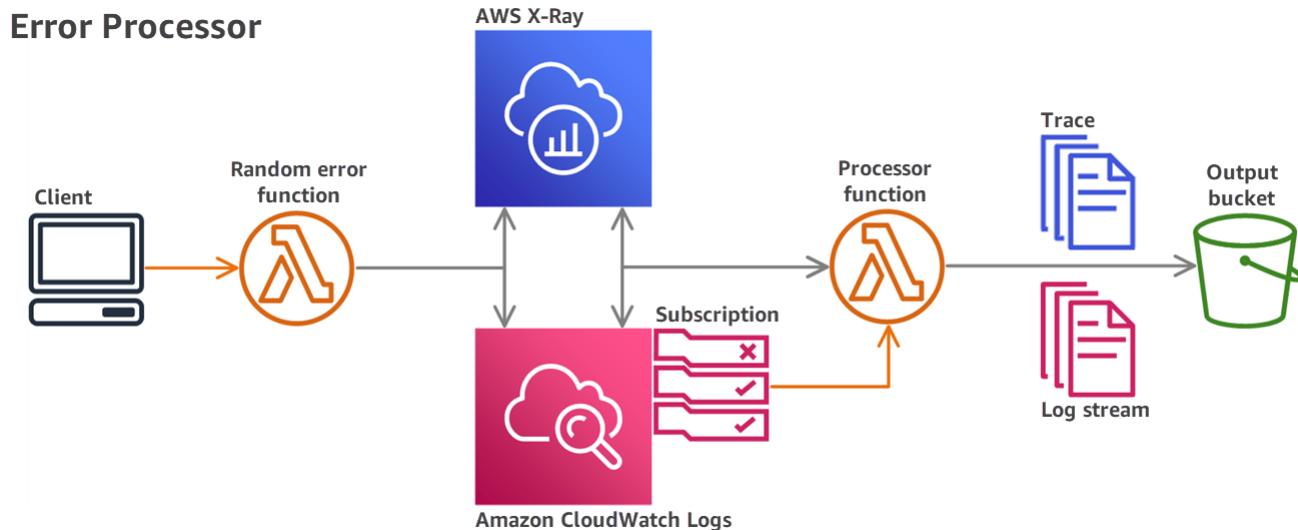
For more information about AWS SAM templates, see [AWS SAM Template Basics](#).

For more information about locally testing Lambda functions, see [Testing and Debugging Serverless Applications](#).

For more information about automating deployments of serverless applications, see [Deploying Serverless Applications](#).

Error Processor Sample Application for AWS Lambda

The Error Processor sample application demonstrates the use of AWS Lambda to handle events from an [Amazon CloudWatch Logs subscription \(p. 164\)](#). CloudWatch Logs lets you invoke a Lambda function when a log entry matches a pattern. The subscription in this application monitors the log group of a function for entries that contain the word `ERROR`. It invokes a processor Lambda function in response. The processor function retrieves the full log stream and trace data for the request that caused the error, and stores them for later use.



Function code is available in the following files:

- Random error – [random-error/index.js](#)
- Processor – [processor/index.js](#)

You can deploy the sample in a few minutes with the AWS CLI and AWS CloudFormation. Follow the instructions in the [README](#) to download, configure, and deploy it in your account.

Sections

- [Architecture and Event Structure \(p. 120\)](#)
- [Instrumentation with AWS X-Ray \(p. 121\)](#)
- [AWS CloudFormation Template and Additional Resources \(p. 122\)](#)

Architecture and Event Structure

The sample application uses the following AWS services.

- AWS Lambda – Runs function code, sends logs to CloudWatch Logs, and sends trace data to X-Ray.
- Amazon CloudWatch Logs – Collects logs, and invokes a function when a log entry matches a filter pattern.
- AWS X-Ray – Collects trace data, indexes traces for search, and generates a service map.
- Amazon Simple Storage Service (Amazon S3) – Stores deployment artifacts and application output.
- AWS CloudFormation – Creates application resources and deploys function code.

A Lambda function in the application generates errors randomly. When CloudWatch Logs detects the word `ERROR` in the function's logs, it sends an event to the processor function for processing.

Example – CloudWatch Logs Message Event

```
{  
  "awslogs": {  
    "data": "H4sIAAAAAAAHWQT0/DMAzFv0vEkbLYcdJkt4qVXmCDteIAm1DbZKjs  
+kdpB0Jo350MhsQFyVLsZ+unl/fJWjeO5asrPgbH5..."  
  }  
}
```

When it's decoded, the data contains details about the log event. The function uses these details to identify the log stream, and parses the log message to get the ID of the request that caused the error.

Example – Decoded CloudWatch Logs Event Data

```
{  
  "messageType": "DATA_MESSAGE",  
  "owner": "123456789012",  
  "logGroup": "/aws/lambda/lambda-error-processor-randomerror-1GD4SSDNACNP4",  
  "logStream": "2019/04/04/[$LATEST]63311769a9d742f19cedf8d2e38995b9",  
  "subscriptionFilters": [  
    "lambda-error-processor-subscription-150PDVQ59CG07"  
  ],  
  "logEvents": [  
    {"id": "34664632210239891980253245280462376874059932423703429141",  
  ]  
}
```

```

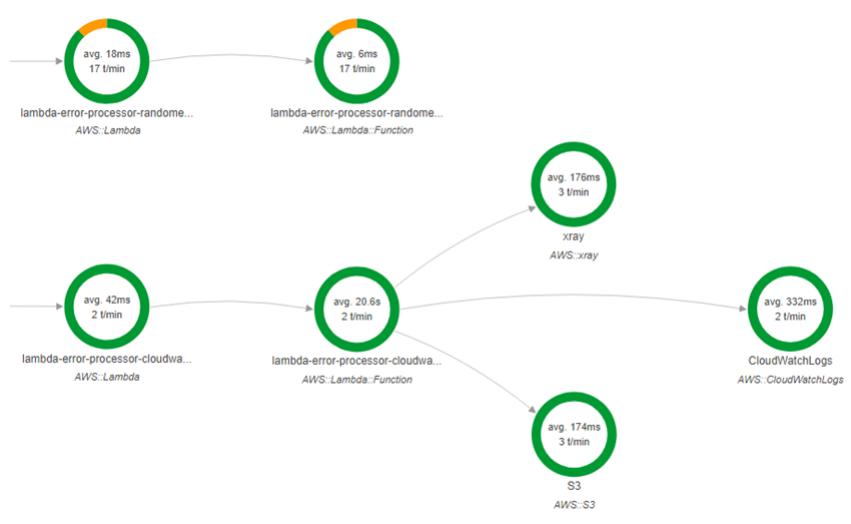
        "timestamp": 1554415868243,
        "message": "2019-04-04T22:11:08.243Z\tt1d2c1444-efd1-43ec-b16e-8fb2d37508b8\terror\n"
    }
}
}

```

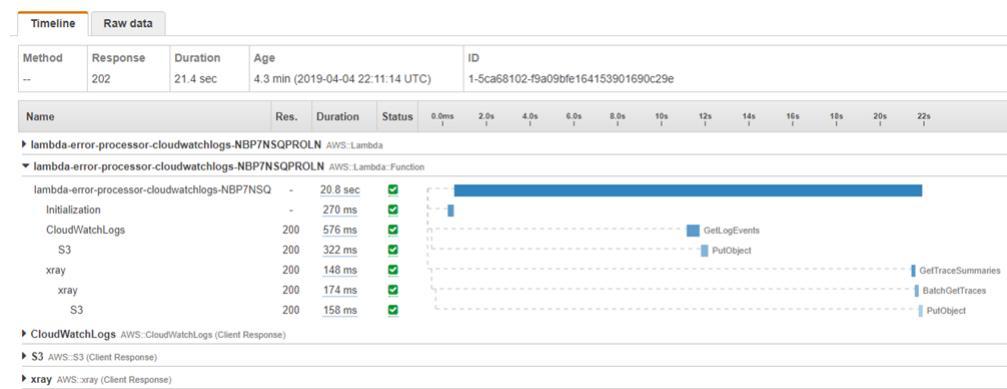
The processor function uses information from the CloudWatch Logs event to download the full log stream and X-Ray trace for a request that caused an error. It stores both in an Amazon S3 bucket. To allow the log stream and trace time to finalize, the function waits for a short period of time before accessing the data.

Instrumentation with AWS X-Ray

The application uses [AWS X-Ray \(p. 232\)](#) to trace function invocations and the calls that functions make to AWS services. X-Ray uses the trace data that it receives from functions to create a service map that helps you identify errors. The following service map shows the random error function generating errors for some requests. It also shows the processor function calling X-Ray, CloudWatch Logs, and Amazon S3.



The two Node.js functions are configured for active tracing in the template, and are instrumented with the AWS X-Ray SDK for Node.js in code. With active tracing, Lambda tags adds a tracing header to incoming requests and sends a trace with timing details to X-Ray. Additionally, the random error function uses the X-Ray SDK to record the request ID and user information in annotations. The annotations are attached to the trace, and you can use them to locate the trace for a specific request.



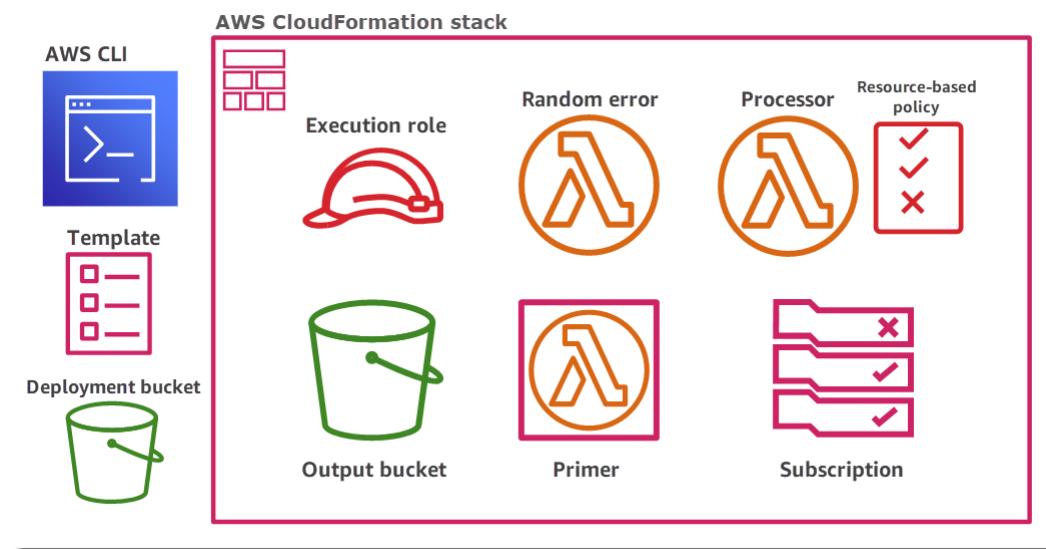
The processor function gets the request ID from the CloudWatch Logs event, and uses the AWS SDK for JavaScript to search X-Ray for that request. It uses AWS SDK clients, which are instrumented with the X-Ray SDK, to download the trace and log stream. Then it stores them in the output bucket. The X-Ray SDK records these calls, and they appear as subsegments in the trace.

AWS CloudFormation Template and Additional Resources

The application is implemented in two Node.js modules—an AWS CloudFormation template and supporting shell scripts. The template creates the processor function, the random error function, and the following supporting resources.

- Execution role – An IAM role that grants the functions permission to access other AWS services.
- Primer function – An additional function that invokes the random error function to create a log group.
- Custom resource – A AWS CloudFormation custom resource that invokes the primer function during deployment to ensure that the log group exists.
- CloudWatch Logs subscription – A subscription for the log stream that triggers the processor function when the word ERROR is logged.
- Resource-based policy – A permission statement on the processor function that allows CloudWatch Logs to invoke it.
- Amazon S3 bucket – A storage location for output from the processor function.

View the template [error-processor.yaml](#) on GitHub.



To work around a limitation of Lambda's integration with AWS CloudFormation, the template creates an additional function that runs during deployments. All Lambda functions come with a CloudWatch Logs log group that stores output from function executions. However, the log group isn't created until the function is invoked for the first time.

To create the subscription, which depends on the existence of the log group, the application uses a third Lambda function to invoke the random error function. The template includes the code for the primer function inline. An AWS CloudFormation custom resource invokes it during deployment. `DependsOn` properties ensure that the log stream and resource-based policy are created prior to the subscription.

Building a Continuous Delivery Pipeline for a Lambda Application with AWS CodePipeline

You can use AWS CodePipeline to create a continuous delivery pipeline for your Lambda application. CodePipeline combines source control, build, and deployment resources to create a pipeline that runs whenever you make a change to your application's source code.

In this tutorial, you create the following resources.

- **Repository** – A Git repository in AWS CodeCommit. When you push a change, the pipeline copies the source code into an Amazon S3 bucket and passes it to the build project.
- **Build project** – An AWS CodeBuild build that gets the source code from the pipeline and packages the application. The source includes a build specification with commands that install dependencies and prepare an AWS Serverless Application Model (AWS SAM) template for deployment.
- **Deployment configuration** – The pipeline's deployment stage defines a set of actions that take the AWS SAM template from the build output, create a change set in AWS CloudFormation, and execute the change set to update the application's AWS CloudFormation stack.
- **Roles** – The pipeline, build, and deployment each have a service role that allows them to manage AWS resources. The console creates the pipeline and build roles when you create those resources. You create the role that allows AWS CloudFormation to manage the application stack.

The pipeline maps a single branch in a repository to a single AWS CloudFormation stack. You can create additional pipelines to add environments for other branches in the same repository. You can also add stages to your pipeline for testing, staging, and manual approvals. For more information about AWS CodePipeline, see [What is AWS CodePipeline](#).

Sections

- [Prerequisites \(p. 123\)](#)
- [Create an AWS CloudFormation Role \(p. 124\)](#)
- [Set Up a Repository \(p. 124\)](#)
- [Create a Pipeline \(p. 126\)](#)
- [Update the Build Stage Role \(p. 127\)](#)
- [Complete the Deployment Stage \(p. 127\)](#)

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Getting Started with AWS Lambda \(p. 3\)](#) to create your first Lambda function.

To follow the procedures in this guide, you will need a command line terminal or shell to run commands. Commands are shown in listings preceded by a prompt symbol (\$) and the name of the current directory, when appropriate:

```
~/lambda-project$ this is a command  
this is output
```

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

On Linux and macOS, use your preferred shell and package manager. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

During the build phase, the build script uploads artifacts to Amazon Simple Storage Service (Amazon S3). You can use an existing bucket, or create a new bucket for the pipeline. Use the AWS CLI to create a bucket.

```
$ aws s3 mb s3://lambda-deployment-artifacts-123456789012
```

Create an AWS CloudFormation Role

Create a role that gives AWS CloudFormation permission to access AWS resources.

To create an AWS CloudFormation role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – AWS CloudFormation**
 - **Permissions – AWSLambdaExecute**
 - **Role name – cfn-lambda-pipeline**
4. Open the role. Under the **Permissions** tab, choose **Add inline policy**.
5. In **Create Policy**, choose the **JSON** tab and add the following policy.

```
{  
    "Statement": [  
        {  
            "Action": [  
                "apigateway:*",  
                "codedeploy:*",  
                "lambda:*",  
                "cloudformation:CreateChangeSet",  
                "iam:GetRole",  
                "iam:CreateRole",  
                "iam:DeleteRole",  
                "iam:PutRolePolicy",  
                "iam:AttachRolePolicy",  
                "iam:DeleteRolePolicy",  
                "iam:DetachRolePolicy",  
                "iam:PassRole",  
                "s3:GetObjectVersion",  
                "s3:GetBucketVersioning"  
            ],  
            "Resource": "*",  
            "Effect": "Allow"  
        }  
    ],  
    "Version": "2012-10-17"  
}
```

Set Up a Repository

Create an AWS CodeCommit repository to store your project files. For more information, see [Setting Up](#) in the CodeCommit User Guide.

To create a repository

1. Open the [Developer Tools console](#).

2. Under **Source**, choose **Repositories**.
3. Choose **Create repository**.
4. Follow the instructions to create and clone a repository named **lambda-pipeline-repo**.

Create the following files in the repository folder.

Example index.js

A Lambda function that returns the current time.

```
var time = require('time');
exports.handler = (event, context, callback) => {
    var currentTime = new time.Date();
    currentTime.setTz("America/Los_Angeles");
    callback(null, {
        statusCode: '200',
        body: 'The time in Los Angeles is: ' + currentTime.toString(),
    });
};
```

Example template.yaml

The [SAM template \(p. 119\)](#) that defines the application.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Outputs the time
Resources:
  TimeFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs8.10
      CodeUri: .
    Events:
      MyTimeApi:
        Type: Api
        Properties:
          Path: /TimeResource
          Method: GET
```

Example buildspec.yml

An [AWS CodeBuild build specification](#) that installs required packages and uploads the deployment package to Amazon S3. Replace the highlighted text with the name of your bucket.

```
version: 0.2
phases:
  install:
    commands:
      - npm install time
      - export BUCKET=λambda-deployment-artifacts-123456789012
      - aws cloudformation package --template-file template.yaml --s3-bucket $BUCKET --
output-template-file outputtemplate.yaml
artifacts:
  type: zip
  files:
    - template.yaml
```

```
- outputtemplate.yaml
```

Commit and push the files to CodeCommit.

```
~/lambda-pipeline-repo$ git add .
~/lambda-pipeline-repo$ git commit -m "project files"
~/lambda-pipeline-repo$ git push
```

Create a Pipeline

Create a pipeline that deploys your application. The pipeline monitors your repository for changes, runs an AWS CodeBuild build to create a deployment package, and deploys the application with AWS CloudFormation. During the pipeline creation process, you also create the AWS CodeBuild build project.

To create a pipeline

1. Open the [Developer Tools console](#).
2. Under **Pipeline**, choose **Pipelines**.
3. Choose **Create pipeline**.
4. Configure the pipeline settings and choose **Next**.
 - **Pipeline name** – `lambda-pipeline`
 - **Service role** – `New service role`
 - **Artifact store** – `Default location`
5. Configure source stage settings and choose **Next**.
 - **Source provider** – `AWS CodeCommit`
 - **Repository name** – `lambda-pipeline-repo`
 - **Branch name** – `master`
 - **Change detection options** – `Amazon CloudWatch Events`
6. For **Build provider**, choose `AWS CodeBuild`, and then choose **Create project**.
7. Configure build project settings and choose **Continue to CodePipeline**.
 - **Project name** – `lambda-pipeline-build`
 - **Operating system** – `Ubuntu`
 - **Runtime** – `Node.js`
 - **Runtime version** – `aws/codebuild/nodejs:8.11.0`
 - **Image version** – `Latest`
 - **Buildspec name** – `buildspec.yml`
8. Choose **Next**.
9. Configure deploy stage settings and choose **Next**.
 - **Deploy provider** – `AWS CloudFormation`
 - **Action mode** – `Create or replace a change set`
 - **Stack name** – `lambda-pipeline-stack`
 - **Change set name** – `lambda-pipeline-changeset`
 - **Template** – `BuildArtifact::outputtemplate.yaml`
 - **Capabilities** – `CAPABILITY_IAM`
 - **Role name** – `cfn-lambda-pipeline`
10. Choose **Create pipeline**.

The pipeline fails the first time it runs because it needs additional permissions. In the next section, you add permissions to the role that's generated for your build stage..

Update the Build Stage Role

During the build stage, AWS CodeBuild needs permission to upload the build output to your Amazon S3 bucket.

To update the role

1. Open the [roles page](#) in the IAM console.
2. Choose **code-build-lambda-pipeline-service-role**.
3. Choose **Attach policies**.
4. Attach **AmazonS3FullAccess**.

Complete the Deployment Stage

The deployment stage has an action that creates a change set for the AWS CloudFormation stack that manages your Lambda application. Add a second action that executes the change set to complete the deployment.

To update the deployment stage

1. Open your pipeline in the [Developer Tools console](#).
2. Choose **Edit**.
3. Next to **Deploy**, choose **Edit stage**.
4. Choose **Add action group**.
5. Configure deploy stage settings and choose **Next**.
 - **Action name – execute-changeset**
 - **Deploy provider – AWS CloudFormation**
 - **Input artifacts – BuildArtifact**
 - **Action mode – Execute a change set**
 - **Stack name – lambda-pipeline-stack**
 - **Change set name – lambda-pipeline-changeset**
6. Choose **Save**.
7. Choose **Done**.
8. Choose **Save**.
9. Choose **Release change** to run the pipeline.

Your pipeline is ready. Push changes to the master branch to trigger a deployment.

Best Practices for Working with AWS Lambda Functions

The following are recommended best practices for using AWS Lambda:

Topics

- [Function Code \(p. 128\)](#)
- [Function Configuration \(p. 129\)](#)
- [Alarming and Metrics \(p. 129\)](#)
- [Stream Event Invokes \(p. 129\)](#)
- [Async Invokes \(p. 130\)](#)
- [Lambda VPC \(p. 130\)](#)

Function Code

- **Separate the Lambda handler (entry point) from your core logic.** This allows you to make a more unit-testable function. In Node.js this may look like:

```
exports.myHandler = function(event, context, callback) {
  var foo = event.foo;
  var bar = event.bar;
  var result = MyLambdaFunction (foo, bar);

  callback(null, result);
}

function MyLambdaFunction (foo, bar) {
  // MyLambdaFunction logic here
}
```

- **Take advantage of Execution Context reuse to improve the performance of your function.** Make sure any externalized configuration or dependencies that your code retrieves are stored and referenced locally after initial execution. Limit the re-initialization of variables/objects on every invocation. Instead use static initialization/constructor, global/static variables and singletons. Keep alive and reuse connections (HTTP, database, etc.) that were established during a previous invocation.
- **Use AWS Lambda Environment Variables (p. 40) to pass operational parameters to your function.** For example, if you are writing to an Amazon S3 bucket, instead of hard-coding the bucket name you are writing to, configure the bucket name as an environment variable.
- **Control the dependencies in your function's deployment package.** The AWS Lambda execution environment contains a number of libraries such as the AWS SDK for the Node.js and Python runtimes (a full list can be found here: [AWS Lambda Runtimes \(p. 102\)](#)). To enable the latest set of features and security updates, Lambda will periodically update these libraries. These updates may introduce subtle changes to the behavior of your Lambda function. To have full control of the dependencies your function uses, we recommend packaging all your dependencies with your deployment package.
- **Minimize your deployment package size to its runtime necessities.** This will reduce the amount of time that it takes for your deployment package to be downloaded and unpacked ahead of invocation. For functions authored in Java or .NET Core, avoid uploading the entire AWS SDK library as part of your deployment package. Instead, selectively depend on the modules which pick up components of the SDK you need (e.g. DynamoDB, Amazon S3 SDK modules and [Lambda core libraries](#)).
- **Reduce the time it takes Lambda to unpack deployment packages** authored in Java by putting your dependency .jar files in a separate /lib directory. This is faster than putting all your function's code in a single jar with a large number of .class files.
- **Minimize the complexity of your dependencies.** Prefer simpler frameworks that load quickly on [Execution Context](#) startup. For example, prefer simpler Java dependency injection (IoC) frameworks like [Dagger](#) or [Guice](#), over more complex ones like [Spring Framework](#).
- **Avoid using recursive code** in your Lambda function, wherein the function automatically calls itself until some arbitrary criteria is met. This could lead to unintended volume of function invocations and escalated costs. If you do accidentally do so, set the function concurrent execution limit to 0 immediately to throttle all invocations to the function, while you update the code.

Function Configuration

- **Performance testing your Lambda function** is a crucial part in ensuring you pick the optimum memory size configuration. Any increase in memory size triggers an equivalent increase in CPU available to your function. The memory usage for your function is determined per-invoke and can be viewed in [AWS CloudWatch Logs](#). On each invoke a `REPORT`: entry will be made, as shown below:

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

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

- **Load test your Lambda function** to determine an optimum timeout value. It is important to analyze how long your function runs so that you can better determine any problems with a dependency service that may increase the concurrency of the function beyond what you expect. This is especially important when your Lambda function makes network calls to resources that may not handle Lambda's scaling.
- **Use most-restrictive permissions when setting IAM policies.** Understand the resources and operations your Lambda function needs, and limit the execution role to these permissions. For more information, see [AWS Lambda Permissions \(p. 9\)](#).
- **Be familiar with AWS Lambda Limits (p. 7).** Payload size, file descriptors and /tmp space are often overlooked when determining runtime resource limits.
- **Delete Lambda functions that you are no longer using.** By doing so, the unused functions won't needlessly count against your deployment package size limit.
- **If you are using Amazon Simple Queue Service** as an event source, make sure the value of the function's expected execution time does not exceed the `Visibility Timeout` value on the queue. This applies both to [CreateFunction \(p. 355\)](#) and [UpdateFunctionConfiguration \(p. 463\)](#).
 - In the case of `CreateFunction`, AWS Lambda will fail the function creation process.
 - In the case of `UpdateFunctionConfiguration`, it could result in duplicate invocations of the function.

Alarming and Metrics

- **Use AWS Lambda Metrics (p. 229) and CloudWatch Alarms** instead of creating or updating a metric from within your Lambda function code. It's a much more efficient way to track the health of your Lambda functions, allowing you to catch issues early in the development process. For instance, you can configure an alarm based on the expected duration of your Lambda function execution time in order to address any bottlenecks or latencies attributable to your function code.
- **Leverage your logging library and AWS Lambda Metrics and Dimensions** to catch app errors (e.g. `ERR`, `ERROR`, `WARNING`, etc.)

Stream Event Invokes

- **Test with different batch and record sizes** so that the polling frequency of each event source is tuned to how quickly your function is able to complete its task. `BatchSize` controls the maximum number of records that can be sent to your function with each invoke. A larger batch size can often more efficiently absorb the invoke overhead across a larger set of records, increasing your throughput.

Note

When there are not enough records to process, instead of waiting, the stream processing function will be invoked with a smaller number of records.

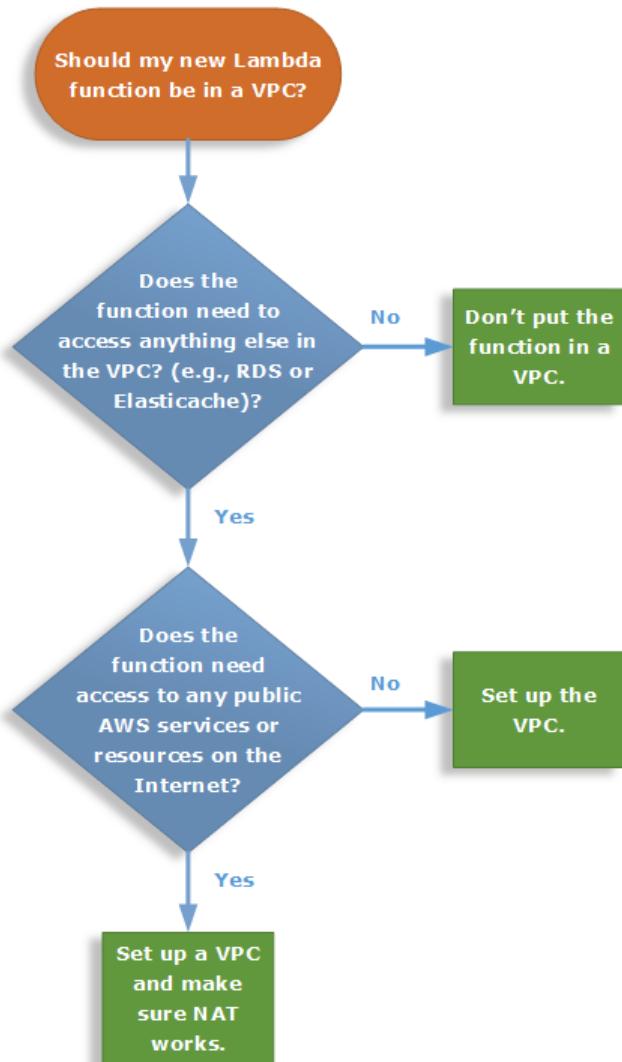
- **Increase Kinesis stream processing throughput by adding shards.** A Kinesis stream is composed of one or more shards. Lambda will poll each shard with at most one concurrent invocation. For example, if your stream has 100 active shards, there will be at most 100 Lambda function invocations running concurrently. Increasing the number of shards will directly increase the number of maximum concurrent Lambda function invocations and can increase your Kinesis stream processing throughput. If you are increasing the number of shards in a Kinesis stream, make sure you have picked a good partition key (see [Partition Keys](#)) for your data, so that related records end up on the same shards and your data is well distributed.
- **Use Amazon CloudWatch** on IteratorAge to determine if your Kinesis stream is being processed. For example, configure a CloudWatch alarm with a maximum setting to 30000 (30 seconds).

Async Invokes

- Create and use [AWS Lambda Function Dead Letter Queues \(p. 88\)](#) to address and replay async function errors.

Lambda VPC

- The following diagram guides you through a decision tree as to whether you should use a VPC (Virtual Private Cloud):



- **Don't put your Lambda function in a VPC unless you have to.** There is no benefit outside of using this to access resources you cannot expose publicly, like a private [Amazon Relational Database](#) instance. Services like Amazon Elasticsearch Service can be secured over IAM with access policies, so exposing the endpoint publicly is safe and wouldn't require you to run your function in the VPC to secure it.
- **Lambda creates elastic network interfaces (ENIs) in your VPC to access your internal resources.** Before requesting a concurrency increase, ensure you have enough ENI capacity (the formula for this can be found here: [Configuring a Lambda Function to Access Resources in an Amazon VPC \(p. 68\)](#)) and IP address space. If you do not have enough ENI capacity, you will need to request an increase. If you do not have enough IP address space, you may need to create a larger subnet.
- **Create dedicated Lambda subnets in your VPC:**
 - This will make it easier to apply a custom route table for NAT Gateway traffic without changing your other private/public subnets. For more information, see [Configuring a Lambda Function to Access Resources in an Amazon VPC \(p. 68\)](#)
 - This also allows you to dedicate an address space to Lambda without sharing it with other resources.

Using AWS Lambda with Other Services

AWS Lambda integrates with other AWS services to invoke functions. You can configure triggers to invoke a function in response to resource lifecycle events, respond to incoming HTTP requests, consume events from a queue, or [run on a schedule \(p. 158\)](#).

Each service that integrates with Lambda sends data to your function in JSON as an event. The structure of the event document is different for each event type, and contains data about the resource or request that triggered the function. Lambda runtimes convert the event into an object and pass it to your function.

The following example shows a test event from an [Application Load Balancer \(p. 133\)](#) that represents a GET request to /lambda?query=1234ABCD.

Example Event from an Application Load Balancer

```
{
  "requestContext": {
    "elb": {
      "targetGroupArn": "arn:aws:elasticloadbalancing:us-east-2:123456789012:targetgroup/lambdata-279XGJDqGZ5rsrHC2Fjr/49e9d65c45c6791a"
    }
  },
  "httpMethod": "GET",
  "path": "/lambda",
  "queryStringParameters": {
    "query": "1234ABCD"
  },
  "headers": {
    "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8",
    "accept-encoding": "gzip",
    "accept-language": "en-US,en;q=0.9",
    "connection": "keep-alive",
    "host": "lambda-alb-123578498.us-east-2.elb.amazonaws.com",
    "upgrade-insecure-requests": "1",
    "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36",
    "x-amzn-trace-id": "Root=1-5c536348-3d683b8b04734faae651f476",
    "x-forwarded-for": "72.12.164.125",
    "x-forwarded-port": "80",
    "x-forwarded-proto": "http",
    "x-imforwards": "20"
  },
  "body": "",
  "isBase64Encoded": false
}
```

Note

The Lambda runtime converts the event document into an object and passes it to your [function handler \(p. 31\)](#). For compiled languages, Lambda provides definitions for event types in a library. See the following topics for more information.

- [Building Lambda Functions with Java \(p. 263\)](#)
- [Building Lambda Functions with Go \(p. 290\)](#)
- [Building Lambda Functions with C# \(p. 302\)](#)

For services that generate a queue or data stream, you create an [event source mapping \(p. 82\)](#) in Lambda and grant Lambda permission to access the other service in the [execution role \(p. 9\)](#). Lambda reads data from the other service, creates an event, and invokes your function.

Services That Lambda Reads Events From

- [Amazon Kinesis \(p. 180\)](#)
- [Amazon DynamoDB \(p. 170\)](#)
- [Amazon Simple Queue Service \(p. 216\)](#)

Other services invoke your function directly. You grant the other service permission in the function's [resource-based policy \(p. 10\)](#), and configure the other service to generate events and invoke your function. Depending on the service, the invocation can be synchronous or asynchronous. For synchronous invocation, the other service waits for the response from your function and might [retry on errors \(p. 85\)](#).

Services That Invoke Lambda Functions Synchronously

- [Elastic Load Balancing \(Application Load Balancer\) \(p. 133\)](#)
- [Amazon Cognito \(p. 168\)](#)
- [Amazon Lex \(p. 193\)](#)
- [Amazon Alexa \(p. 135\)](#)
- [Amazon API Gateway \(p. 135\)](#)
- [Amazon CloudFront \(Lambda@Edge\) \(p. 166\)](#)
- [Amazon Kinesis Data Firehose \(p. 192\)](#)

For asynchronous invocation, Lambda queues the event before passing it to your function. The other service gets a success response as soon as the event is queued and isn't aware of what happens afterwards. If an error occurs, Lambda handles [retries \(p. 85\)](#), and can send failed events to a [dead-letter queue \(p. 88\)](#) that you configure.

Services That Invoke Lambda Functions Asynchronously

- [Amazon Simple Storage Service \(p. 194\)](#)
- [Amazon Simple Notification Service \(p. 211\)](#)
- [Amazon Simple Email Service \(p. 209\)](#)
- [AWS CloudFormation \(p. 164\)](#)
- [Amazon CloudWatch Logs \(p. 164\)](#)
- [Amazon CloudWatch Events \(p. 158\)](#)
- [AWS CodeCommit \(p. 168\)](#)
- [AWS Config \(p. 169\)](#)

See the topics in this chapter for more details about each service, and example events that you can use to test your function.

Using AWS Lambda with an Application Load Balancer

You can use a Lambda function to process requests from an Application Load Balancer. Elastic Load Balancing supports Lambda functions as a target for an Application Load Balancer. Use load balancer

rules to route HTTP requests to a function, based on path or header values. Process the request and return an HTTP response from your Lambda function.

Elastic Load Balancing invokes your Lambda function synchronously with an event that contains the request body and metadata.

Example Application Load Balancer Request Event

```
{  
    "requestContext": {  
        "elb": {  
            "targetGroupArn": "arn:aws:elasticloadbalancing:us-east-2:123456789012:targetgroup/lambdam-279XGJDqGZ5rsrHC2FJr/49e9d65c45c6791a"  
        }  
    },  
    "httpMethod": "GET",  
    "path": "/lambda",  
    "queryStringParameters": {  
        "query": "1234ABCD"  
    },  
    "headers": {  
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8",  
        "accept-encoding": "gzip",  
        "accept-language": "en-US,en;q=0.9",  
        "connection": "keep-alive",  
        "host": "lambda-alb-123456789012.us-east-2.elb.amazonaws.com",  
        "upgrade-insecure-requests": "1",  
        "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36",  
        "x-amzn-trace-id": "Root=1-5c36348-3d683b8b04734faae651f476",  
        "x-forwarded-for": "72.12.164.125",  
        "x-forwarded-port": "80",  
        "x-forwarded-proto": "http",  
        "x-imforwards": "20"  
    },  
    "body": "",  
    "isBase64Encoded": false  
}
```

Your function processes the event and returns a response to the load balancer in JSON. Elastic Load Balancing converts the response to HTTP and returns it to the user.

Example Response Format

```
{  
    "statusCode": 200,  
    "statusDescription": "HTTP OK",  
    "isBase64Encoded": False,  
    "headers": {  
        "Content-Type": "text/html"  
    },  
    "body": "<h1>Hello from Lambda!</h1>"  
}
```

To configure an Application Load Balancer as a function trigger, grant Elastic Load Balancing permission to execute the function, create a target group that routes requests to the function, and add a rule to the load balancer that sends requests to the target group.

Use the `add-permission` command to add a permission statement to your function's resource-based policy.

```
$ aws lambda add-permission --function-name alb-function \
--statement-id load-balancer --action "lambda:InvokeFunction" \
--principal elasticloadbalancing.amazonaws.com
{
    "Statement": "{\"Sid\":\"load-balancer\",\"Effect\":\"Allow\",\"Principal\":{\"Service\":\"elasticloadbalancing.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",\"Resource\":\"arn:aws:lambda:us-west-2:123456789012:function:alb-function\"}"
}
```

For instructions on configuring the Application Load Balancer listener and target group, see [Lambda Functions as a Target](#) in the *User Guide for Application Load Balancers*.

Using AWS Lambda with Alexa

You can use Lambda functions to build services that give new skills to Alexa, the Voice assistant on Amazon Echo. The Alexa Skills Kit provides the APIs, tools, and documentation to create these new skills, powered by your own services running as Lambda functions. Amazon Echo users can access these new skills by asking Alexa questions or making requests.

The Alexa Skills Kit is available on GitHub.

- [Alexa Skills Kit SDK for Node.js](#)
- [Alexa Skills Kit SDK for Java](#)

Example Alexa Smart Home Event

```
{
  "header": {
    "payloadVersion": "1",
    "namespace": "Control",
    "name": "SwitchOnOffRequest"
  },
  "payload": {
    "switchControlAction": "TURN_ON",
    "appliance": {
      "additionalApplianceDetails": {
        "key2": "value2",
        "key1": "value1"
      },
      "applianceId": "sampleId"
    },
    "accessToken": "sampleAccessToken"
  }
}
```

For more information, see [Getting Started with Alexa Skills Kit](#).

Using AWS Lambda with Amazon API Gateway

You can invoke AWS Lambda functions over HTTPS. You can do this by defining a custom REST API and endpoint using [Amazon API Gateway](#), and then mapping individual methods, such as `GET` and `PUT`, to specific Lambda functions. Alternatively, you could add a special method named `ANY` to map all supported methods (`GET`, `POST`, `PATCH`, `DELETE`) to your Lambda function. When you send an HTTPS request to the API endpoint, the Amazon API Gateway service invokes the corresponding Lambda function. For more information about the `ANY` method, see [Create a Simple Microservice using Lambda and API Gateway \(p. 148\)](#).

Example Amazon API Gateway Message Event

```
{
  "path": "/test/hello",
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8",
    "Accept-Encoding": "gzip, deflate, lzma, sdch, br",
    "Accept-Language": "en-US,en;q=0.8",
    "CloudFront-Forwarded-Proto": "https",
    "CloudFront-Is-Desktop-Viewer": "true",
    "CloudFront-Is-Mobile-Viewer": "false",
    "CloudFront-Is-SmartTV-Viewer": "false",
    "CloudFront-Is-Tablet-Viewer": "false",
    "CloudFront-Viewer-Country": "US",
    "Host": "wt6mne2s9k.execute-api.us-west-2.amazonaws.com",
    "Upgrade-Insecure-Requests": "1",
    "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.82 Safari/537.36 OPR/39.0.2256.48",
    "Via": "1.1 fb7cca60f0ecd82ce07790c9c5eef16c.cloudfront.net (CloudFront)",
    "X-Amz-Cf-Id": "nBsWBOOrSHMgnaROZJK1wGCZ9PcRcSpq_oSXZNQwQ100TZL4cimZo3g==",
    "X-Forwarded-For": "192.168.100.1, 192.168.1.1",
    "X-Forwarded-Port": "443",
    "X-Forwarded-Proto": "https"
  },
  "pathParameters": {
    "proxy": "hello"
  },
  "requestContext": {
    "accountId": "123456789012",
    "resourceId": "us4z18",
    "stage": "test",
    "requestId": "41b45ea3-70b5-11e6-b7bd-69b5aaebc7d9",
    "identity": {
      "cognitoIdentityPoolId": "",
      "accountId": "",
      "cognitoIdentityId": "",
      "caller": "",
      "apiKey": "",
      "sourceIp": "192.168.100.1",
      "cognitoAuthenticationType": "",
      "cognitoAuthenticationProvider": "",
      "userArn": "",
      "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.82 Safari/537.36 OPR/39.0.2256.48",
      "user": ""
    },
    "resourcePath": "/{proxy+}",
    "httpMethod": "GET",
    "apiId": "wt6mne2s9k"
  },
  "resource": "/{proxy+}",
  "httpMethod": "GET",
  "queryStringParameters": {
    "name": "me"
  },
  "stageVariables": {
    "stageVarName": "stageVarValue"
  }
}
```

Amazon API Gateway also adds a layer between your application users and your app logic that enables the following:

- Ability to throttle individual users or requests.

- Protect against Distributed Denial of Service attacks.
- Provide a caching layer to cache response from your Lambda function.

Note the following about how the Amazon API Gateway and AWS Lambda integration works:

- **Push-event model** – This is a model (see [AWS Lambda Event Source Mapping \(p. 82\)](#)), where Amazon API Gateway invokes the Lambda function by passing data in the request body as parameter to the Lambda function.
- **Synchronous invocation** – The Amazon API Gateway can invoke the Lambda function and get a response back in real time by specifying RequestResponse as the invocation type. For information about invocation types, see [Invocation Types \(p. 82\)](#).
- **Event structure** – The event your Lambda function receives is the body from the HTTPS request that Amazon API Gateway receives and your Lambda function is the custom code written to process the specific event type.

Note that there are two types of permissions policies that you work with when you set up the end-to-end experience:

- **Permissions for your Lambda function** – Regardless of what invokes a Lambda function, AWS Lambda executes the function by assuming the IAM role (execution role) that you specify at the time you create the Lambda function. Using the permissions policy associated with this role, you grant your Lambda function the permissions that it needs. For example, if your Lambda function needs to read an object, you grant permissions for the relevant Amazon S3 actions in the permissions policy. For more information, see [AWS Lambda Execution Role \(p. 9\)](#).
- **Permission for Amazon API Gateway to invoke your Lambda function** – Amazon API Gateway cannot invoke your Lambda function without your permission. You grant this permission via the permission policy associated with the Lambda function.

Tutorial: Using AWS Lambda with Amazon API Gateway

In this example you create a simple API using Amazon API Gateway. An Amazon API Gateway is a collection of resources and methods. For this tutorial, you create one resource (`DynamoDBManager`) and define one method (`POST`) on it. The method is backed by a Lambda function (`LambdaFunctionOverHttps`). That is, when you call the API through an HTTPS endpoint, Amazon API Gateway invokes the Lambda function.

The `POST` method on the `DynamoDBManager` resource supports the following DynamoDB operations:

- Create, update, and delete an item.
- Read an item.
- Scan an item.
- Other operations (echo, ping), not related to DynamoDB, that you can use for testing.

The request payload you send in the `POST` request identifies the DynamoDB operation and provides necessary data. For example:

- The following is a sample request payload for a DynamoDB create item operation:

```
{  
    "operation": "create",  
    "tableName": "lambda-apigateway",  
}
```

```
    "payload": {
        "Item": {
            "id": "1",
            "name": "Bob"
        }
    }
}
```

- The following is a sample request payload for a DynamoDB read item operation:

```
{
    "operation": "read",
    "tableName": "lambda-apigateway",
    "payload": {
        "Key": {
            "id": "1"
        }
    }
}
```

- The following is a sample request payload for an echo operation. You send an HTTP POST request to the endpoint, using the following data in the request body.

```
{
    "operation": "echo",
    "payload": {
        "somekey1": "somevalue1",
        "somekey2": "somevalue2"
    }
}
```

Note

API Gateway offers advanced capabilities, such as:

- **Pass through the entire request** – A Lambda function can receive the entire HTTP request (instead of just the request body) and set the HTTP response (instead of just the response body) using the AWS_PROXY integration type.
- **Catch-all methods** – Map all methods of an API resource to a single Lambda function with a single mapping, using the ANY catch-all method.
- **Catch-all resources** – Map all sub-paths of a resource to a Lambda function without any additional configuration using the new path parameter ({proxy+}).

To learn more about these API Gateway features, see [Configure Proxy Integration for a Proxy Resource](#).

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Getting Started with AWS Lambda \(p. 3\)](#) to create your first Lambda function.

To follow the procedures in this guide, you will need a command line terminal or shell to run commands. Commands are shown in listings preceded by a prompt symbol (\$) and the name of the current directory, when appropriate:

```
~/lambda-project$ this is a command
this is output
```

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

On Linux and macOS, use your preferred shell and package manager. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the Execution Role

Create the [execution role \(p. 9\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity** – Lambda.
 - **Role name** – **lambda-apigateway-role**.
 - **Permissions** – Custom policy with permission to DynamoDB and CloudWatch Logs.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "Stmt1428341300017",  
            "Action": [  
                "dynamodb>DeleteItem",  
                "dynamodb>GetItem",  
                "dynamodb>PutItem",  
                "dynamodb>Query",  
                "dynamodb>Scan",  
                "dynamodb>UpdateItem"  
            ],  
            "Effect": "Allow",  
            "Resource": "*"  
        },  
        {  
            "Sid": "",  
            "Resource": "*",  
            "Action": [  
                "logs>CreateLogGroup",  
                "logs>CreateLogStream",  
                "logs>PutLogEvents"  
            ],  
            "Effect": "Allow"  
        }  
    ]  
}
```

The custom policy has the permissions that the function needs to write data to DynamoDB and upload logs. Note the Amazon Resource Name (ARN) of the role for later use.

Create the Function

The following example code receives a Kinesis event input and processes the messages that it contains. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Note

For sample code in other languages, see [Sample Function Code \(p. 146\)](#).

Example index.js

```
console.log('Loading function');

var AWS = require('aws-sdk');
var dynamo = new AWS.DynamoDB.DocumentClient();

/**
 * Provide an event that contains the following keys:
 *
 * - operation: one of the operations in the switch statement below
 * - tableName: required for operations that interact with DynamoDB
 * - payload: a parameter to pass to the operation being performed
 */
exports.handler = function(event, context, callback) {
    //console.log('Received event:', JSON.stringify(event, null, 2));

    var operation = event.operation;

    if (event.tableName) {
        event.payload.TableName = event.tableName;
    }

    switch (operation) {
        case 'create':
            dynamo.put(event.payload, callback);
            break;
        case 'read':
            dynamo.get(event.payload, callback);
            break;
        case 'update':
            dynamo.update(event.payload, callback);
            break;
        case 'delete':
            dynamo.delete(event.payload, callback);
            break;
        case 'list':
            dynamo.scan(event.payload, callback);
            break;
        case 'echo':
            callback(null, "Success");
            break;
        case 'ping':
            callback(null, "pong");
            break;
        default:
            callback('Unknown operation: ${operation}');
    }
};


```

To create the function

1. Copy the sample code into a file named `index.js`.
2. Create a deployment package.

```
$ zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command.

```
$ aws lambda create-function --function-name LambdaFunctionOverHttps \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs8.10 \
--role arn:aws:iam::123456789012:role/service-role/lambda-apigateway-role
```

Test the Lambda Function

Invoke the function manually using the sample event data. We recommend that you invoke the function using the console because the console UI provides a user-friendly interface for reviewing the execution results, including the execution summary, logs written by your code, and the results returned by the function (because the console always performs synchronous execution—invokes the Lambda function using the `RequestResponse` invocation type).

To test the Lambda function

1. Copy the following JSON into a file and save it as `input.txt`.

```
{  
    "operation": "echo",  
    "payload": {  
        "somekey1": "somevalue1",  
        "somekey2": "somevalue2"  
    }  
}
```

2. Execute the following `invoke` command:

```
$ aws lambda invoke --function-name LambdaFunctionOverHttps \  
--payload fileb://input.txt outfile.txt
```

Create an API Using Amazon API Gateway

In this step, you associate your Lambda function with a method in the API that you created using Amazon API Gateway and test the end-to-end experience. That is, when an HTTP request is sent to an API method, Amazon API Gateway invokes your Lambda function.

First, you create an API (`DynamoDBOperations`) using Amazon API Gateway with one resource (`DynamoDBManager`) and one method (`POST`). You associate the `POST` method with your Lambda function. Then, you test the end-to-end experience.

Create the API

Run the following `create-rest-api` command to create the `DynamoDBOperations` API for this tutorial.

```
$ aws apigateway create-rest-api --name DynamoDBOperations  
{  
    "id": "bs8fqo6bp0",  
    "name": "DynamoDBOperations",  
    "createdDate": 1539803980,  
    "apiKeySource": "HEADER",  
    "endpointConfiguration": {  
        "types": [  
            "EDGE"  
        ]  
    }  
}
```

Save the API ID for use in further commands. You also need the ID of the API root resource. To get the ID, run the `get-resources` command.

```
$ API=bs8fqo6bp0
```

```
$ aws apigateway get-resources --rest-api-id $API
{
    "items": [
        {
            "path": "/",
            "id": "e8kitthgdb"
        }
    ]
}
```

At this time you only have the root resource, but you add more resources in the next step.

Create a Resource in the API

Run the following `create-resource` command to create a resource (`DynamoDBManager`) in the API that you created in the preceding section.

```
$ aws apigateway create-resource --rest-api-id $API --path-part DynamoDBManager \
--parent-id e8kitthgdb
{
    "path": "/DynamoDBManager",
    "pathPart": "DynamoDBManager",
    "id": "resource-id",
    "parentId": "e8kitthgdb"
}
```

Note the ID in the response. This is the ID of the `DynamoDBManager` resource that you created.

Create POST Method on the Resource

Run the following `put-method` command to create a `POST` method on the `DynamoDBManager` resource in your API.

```
$ RESOURCE=iuig5w
$ aws apigateway put-method --rest-api-id $API --resource-id $RESOURCE \
--http-method POST --authorization-type NONE
{
    "apiKeyRequired": false,
    "httpMethod": "POST",
    "authorizationType": "NONE"
}
```

We specify `NONE` for the `--authorization-type` parameter, which means that unauthenticated requests for this method are supported. This is fine for testing but in production you should use either the key-based or role-base authentication.

Set the Lambda Function as the Destination for the POST Method

Run the following command to set the Lambda function as the integration point for the `POST` method. This is the method Amazon API Gateway invokes when you make an HTTP request for the `POST` method endpoint. This command and others use ARNs that include your account ID and region. Save these to variables (you can find your account ID in the role ARN that you used to create the function).

```
$ REGION=us-east-2
$ ACCOUNT=123456789012
$ aws apigateway put-integration --rest-api-id $API --resource-id $RESOURCE \
--http-method POST --type AWS --integration-http-method POST \
--uri arn:aws:apigateway:$REGION:lambda:path/2015-03-31/functions/arn:aws:lambda:$REGION:
$ACCOUNT:function:LambdaFunctionOverHttps/invocations
{
```

```
    "type": "AWS",
    "httpMethod": "POST",
    "uri": "arn:aws:apigateway:us-east-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-2:123456789012:function:LambdaFunctionOverHttps/invocations",
    "passthroughBehavior": "WHEN_NO_MATCH",
    "timeoutInMillis": 29000,
    "cacheNamespace": "iuig5w",
    "cacheKeyParameters": []
}
```

--integration-http-method is the method that API Gateway uses to communicate with AWS Lambda. --uri is unique identifier for the endpoint to which Amazon API Gateway can send request.

Set content-type of the POST method response and integration response to JSON as follows:

- Run the following command to set the POST method response to JSON. This is the response type that your API method returns.

```
$ aws apigateway put-method-response --rest-api-id $API \
--resource-id $RESOURCE --http-method POST \
--status-code 200 --response-models application/json=Empty
{
    "statusCode": "200",
    "responseModels": {
        "application/json": "Empty"
    }
}
```

- Run the following command to set the POST method integration response to JSON. This is the response type that Lambda function returns.

```
$ aws apigateway put-integration-response --rest-api-id $API \
--resource-id $RESOURCE --http-method POST \
--status-code 200 --response-templates application/json=""
{
    "statusCode": "200",
    "responseTemplates": {
        "application/json": null
    }
}
```

Deploy the API

In this step, you deploy the API that you created to a stage called prod.

```
$ aws apigateway create-deployment --rest-api-id $API --stage-name prod
{
    "id": "20vgsz",
    "createdDate": 1539820012
}
```

Grant Invoke Permission to the API

Now that you have an API created using Amazon API Gateway and you've deployed it, you can test. First, you need to add permissions so that Amazon API Gateway can invoke your Lambda function when you send HTTP request to the POST method.

To do this, you need to add a permissions to the permissions policy associated with your Lambda function. Run the following add-permission AWS Lambda command to grant the Amazon API

Gateway service principal (`apigateway.amazonaws.com`) permissions to invoke your Lambda function (`LambdaFunctionOverHttps`).

```
$ aws lambda add-permission --function-name LambdaFunctionOverHttps \
--statement-id apigateway-test-2 --action lambda:InvokeFunction \
--principal apigateway.amazonaws.com \
--source-arn "arn:aws:execute-api:$REGION:$ACCOUNT:$API/*/POST/DynamoDBManager"
{
    "Statement": "{\"Sid\":\"apigateway-test-2\",\"Effect\":\"Allow\",\"Principal\":
    \"Service\":\"apigateway.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",
    \"Resource\":\"arn:aws:lambda:us-east-2:123456789012:function:LambdaFunctionOverHttps\",
    \"Condition\":\"{\\\"ArnLike\\\":{\\\"AWS:SourceArn\\\":\\\"arn:aws:execute-api:us-east-2:123456789012:mnh1yprki7/*
    POST/DynamoDBManager\\\"}}\""
}
```

You must grant this permission to enable testing (if you go to the Amazon API Gateway and choose **Test** to test the API method, you need this permission). Note the `--source-arn` specifies a wildcard character (*) as the stage value (indicates testing only). This allows you to test without deploying the API.

Now, run the same command again, but this time you grant to your deployed API permissions to invoke the Lambda function.

```
$ aws lambda add-permission --function-name LambdaFunctionOverHttps \
--statement-id apigateway-prod-2 --action lambda:InvokeFunction \
--principal apigateway.amazonaws.com \
--source-arn "arn:aws:execute-api:$REGION:$ACCOUNT:$API/prod/POST/DynamoDBManager"
{
    "Statement": "{\"Sid\":\"apigateway-prod-2\",\"Effect\":\"Allow\",\"Principal\":
    \"Service\":\"apigateway.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",
    \"Resource\":\"arn:aws:lambda:us-east-2:123456789012:function:LambdaFunctionOverHttps\",
    \"Condition\":\"{\\\"ArnLike\\\":{\\\"AWS:SourceArn\\\":\\\"arn:aws:execute-api:us-east-2:123456789012:mnh1yprki7/
    prod/POST/DynamoDBManager\\\"}}\""
}
```

You grant this permission so that your deployed API has permissions to invoke the Lambda function. Note that the `--source-arn` specifies a `prod` which is the stage name we used when deploying the API.

Create a Amazon DynamoDB Table

Create the DynamoDB table that the Lambda function uses.

To create a DynamoDB table

1. Open the [DynamoDB console](#).
2. Choose **Create table**.
3. Create a table with the following settings.
 - **Table name** – `lambda-apigateway`
 - **Primary key** – `id` (string)
4. Choose **Create**.

Trigger the Function with an HTTP Request

In this step, you are ready to send an HTTP request to the `POST` method endpoint. You can use either Curl or a method (`test-invoke-method`) provided by Amazon API Gateway.

You can use Amazon API Gateway CLI commands to send an HTTP POST request to the resource (`DynamoDBManager`) endpoint. Because you deployed your Amazon API Gateway, you can use Curl to invoke the methods for the same operation.

The Lambda function supports using the `create` operation to create an item in your DynamoDB table. To request this operation, use the following JSON:

Example `create-item.json`

```
{  
    "operation": "create",  
    "tableName": "lambda-apigateway",  
    "payload": {  
        "Item": {  
            "id": "1234ABCD",  
            "number": 5  
        }  
    }  
}
```

Save the test input to a file named `create-item.json`. Run the `test-invoke-method` Amazon API Gateway command to send an HTTP POST method request to the resource (`DynamoDBManager`) endpoint.

```
$ aws apigateway test-invoke-method --rest-api-id $API \  
--resource-id $RESOURCE --http-method POST --path-with-query-string "" \  
--body file://create-item.json
```

Or, you can use the following Curl command:

```
$ curl -X POST -d "{\"operation\":\"create\", \"tableName\":\"lambda-apigateway\",  
\"payload\":{\"Item\":{\"id\":\"1\", \"name\":\"Bob\"}}}" https://$API.execute-api.  
$REGION.amazonaws.com/prod/DynamoDBManager
```

To send request for the echo operation that your Lambda function supports, you can use the following request payload:

Example `echo.json`

```
{  
    "operation": "echo",  
    "payload": {  
        "somekey1": "somevalue1",  
        "somekey2": "somevalue2"  
    }  
}
```

Save the test input to a file named `echo.json`. Run the `test-invoke-method` Amazon API Gateway CLI command to send an HTTP POST method request to the resource (`DynamoDBManager`) endpoint using the preceding JSON in the request body.

```
$ aws apigateway test-invoke-method --rest-api-id $API \  
--resource-id $RESOURCE --http-method POST --path-with-query-string "" \  
--body file://echo.json
```

Or, you can use the following Curl command:

```
$ curl -X POST -d "{\"operation\":\"echo\",\"payload\":{\"somekey1\":\"somevalue1\", \"somekey2\":\"somevalue2\"}}" https://$API.execute-api.$REGION.amazonaws.com/prod/DynamoDBManager
```

Sample Function Code

Sample code is available for the following languages.

Topics

- [Node.js \(p. 146\)](#)
- [Python 3 \(p. 147\)](#)
- [Go \(p. 147\)](#)

Node.js

The following example processes messages from API Gateway, and manages DynamoDB documents based on the request method.

Example index.js

```
console.log('Loading function');

var AWS = require('aws-sdk');
var dynamo = new AWS.DynamoDB.DocumentClient();

/**
 * Provide an event that contains the following keys:
 *
 * - operation: one of the operations in the switch statement below
 * - tableName: required for operations that interact with DynamoDB
 * - payload: a parameter to pass to the operation being performed
 */
exports.handler = function(event, context, callback) {
    //console.log('Received event:', JSON.stringify(event, null, 2));

    var operation = event.operation;

    if (event.tableName) {
        event.payload.TableName = event.tableName;
    }

    switch (operation) {
        case 'create':
            dynamo.put(event.payload, callback);
            break;
        case 'read':
            dynamo.get(event.payload, callback);
            break;
        case 'update':
            dynamo.update(event.payload, callback);
            break;
        case 'delete':
            dynamo.delete(event.payload, callback);
            break;
        case 'list':
            dynamo.scan(event.payload, callback);
            break;
        case 'echo':
            callback(null, "Success");
    }
}
```

```
        break;
    case 'ping':
        callback(null, "pong");
        break;
    default:
        callback('Unknown operation: ${operation}');
}
};
```

Zip up the sample code to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Node.js \(p. 241\)](#).

Python 3

The following example processes messages from API Gateway, and manages DynamoDB documents based on the request method.

Example LambdaFunctionOverHttps.py

```
from __future__ import print_function

import boto3
import json

print('Loading function')


def handler(event, context):
    '''Provide an event that contains the following keys:
       - operation: one of the operations in the operations dict below
       - tableName: required for operations that interact with DynamoDB
       - payload: a parameter to pass to the operation being performed
    '''
    #print("Received event: " + json.dumps(event, indent=2))

    operation = event['operation']

    if 'tableName' in event:
        dynamo = boto3.resource('dynamodb').Table(event['tableName'])

    operations = {
        'create': lambda x: dynamo.put_item(**x),
        'read': lambda x: dynamo.get_item(**x),
        'update': lambda x: dynamo.update_item(**x),
        'delete': lambda x: dynamo.delete_item(**x),
        'list': lambda x: dynamo.scan(**x),
        'echo': lambda x: x,
        'ping': lambda x: 'pong'
    }

    if operation in operations:
        return operations[operation](event.get('payload'))
    else:
        raise ValueError('Unrecognized operation "{}".format(operation))
```

Zip up the sample code to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Python \(p. 251\)](#).

Go

The following example processes messages from API Gateway, and logs information about the request.

Example LambdaFunctionOverHttps.go

```
import (
    "strings"
    "github.com/aws/aws-lambda-go/events"
)

func handleRequest(ctx context.Context, request events.APIGatewayProxyRequest)
(events.APIGatewayProxyResponse, error) {
    fmt.Printf("Processing request data for request %s.\n",
request.RequestContext.RequestId)
    fmt.Printf("Body size = %d.\n", len(request.Body))

    fmt.Println("Headers:")
    for key, value := range request.Headers {
        fmt.Printf("    %s: %s\n", key, value)
    }

    return events.APIGatewayProxyResponse { Body: request.Body, StatusCode: 200 }, nil
}
```

Build the executable with `go build` and create a deployment package. For instructions, see [AWS Lambda Deployment Package in Go \(p. 290\)](#).

Create a Simple Microservice using Lambda and API Gateway

In this tutorial you will use the Lambda console to create a Lambda function, and an Amazon API Gateway endpoint to trigger that function. You will be able to call the endpoint with any method (GET, POST, PATCH, etc.) to trigger your Lambda function. When the endpoint is called, the entire request will be passed through to your Lambda function. Your function action will depend on the method you call your endpoint with:

- **DELETE**: delete an item from a DynamoDB table
- **GET**: scan table and return all items
- **POST**: Create an item
- **PUT**: Update an item

Create an API Using Amazon API Gateway

Follow the steps in this section to create a new Lambda function and an API Gateway endpoint to trigger it:

To create an API

1. Sign in to the AWS Management Console and open the AWS Lambda console.
2. Choose **Create Lambda function**.
3. Choose **Blueprint**.
4. Enter **microservice** in the search bar. Choose the **microservice-http-endpoint** blueprint and then choose **Configure**.
5. Configure the following settings.
 - **Name** – `lambda-microservice`.
 - **Role** – **Create a new role from one or more templates**.

- **Role name – lambda-apigateway-role.**
- **Policy templates – Simple microservice permissions.**
- **API – Create a new API.**
- **Security – Open.**

Choose **Create function**.

When you complete the wizard and create your function, Lambda creates a proxy resource named `lambda-microservice` under the API name you selected. For more information about proxy resources, see [Configure Proxy Integration for a Proxy Resource](#).

A proxy resource has an `AWS_PROXY` integration type and a catch-all method `ANY`. The `AWS_PROXY` integration type applies a default mapping template to pass through the entire request to the Lambda function and transforms the output from the Lambda function to HTTP responses. The `ANY` method defines the same integration setup for all the supported methods, including `GET`, `POST`, `PATCH`, `DELETE` and others.

Test Sending an HTTPS Request

In this step, you will use the console to test the Lambda function. In addition, you can run a `curl` command to test the end-to-end experience. That is, send an HTTPS request to your API method and have Amazon API Gateway invoke your Lambda function. In order to complete the steps, make sure you have created a DynamoDB table and named it "MyTable". For more information, see [Create a DynamoDB Table with a Stream Enabled \(p. 175\)](#)

To test the API

1. With your `MyLambdaMicroService` function still open in the console, choose the **Actions** tab and then choose **Configure test event**.
2. Replace the existing text with the following:

```
{  
  "httpMethod": "GET",  
  "queryStringParameters": {  
    "TableName": "MyTable"  
  }  
}
```

3. After entering the text above choose **Save and test**.

AWS SAM Template for an API Gateway Application

You can build this application using [AWS SAM](#). To learn more about creating AWS SAM templates, see [AWS SAM Template Basics](#) in the [AWS Serverless Application Model Developer Guide](#).

Below is a sample AWS SAM template for the Lambda application from the [tutorial \(p. 137\)](#). Copy the text below to a `.yaml` file and save it next to the ZIP package you created previously. Note that the `Handler` and `Runtime` parameter values should match the ones you used when you created the function in the previous section.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'  
Transform: AWS::Serverless-2016-10-31
```

```
Resources:
  LambdaFunctionOverHttps:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs8.10
      Policies: AmazonDynamoDBFullAccess
    Events:
      HttpPost:
        Type: Api
        Properties:
          Path: '/DynamoDBOperations/DynamoDBManager'
          Method: post
```

For information on how to package and deploy your serverless application using the package and deploy commands, see [Deploying Serverless Applications](#) in the *AWS Serverless Application Model Developer Guide*.

Using AWS Lambda with AWS CloudTrail

AWS CloudTrail is a service that provides a record of actions taken by a user, role, or an AWS service. CloudTrail captures API calls as events. For an ongoing record of events in your AWS account, you create a trail. A trail enables CloudTrail to deliver log files of events to an Amazon S3 bucket.

You can take advantage of Amazon S3's bucket notification feature and direct Amazon S3 to publish object-created events to AWS Lambda. Whenever CloudTrail writes logs to your S3 bucket, Amazon S3 can then invoke your Lambda function by passing the Amazon S3 object-created event as a parameter. The S3 event provides information, including the bucket name and key name of the log object that CloudTrail created. Your Lambda function code can read the log object and process the access records logged by CloudTrail. For example, you might write Lambda function code to notify you if specific API call was made in your account.

In this scenario, CloudTrail writes access logs to your S3 bucket. As for AWS Lambda, Amazon S3 is the event source so Amazon S3 publishes events to AWS Lambda and invokes your Lambda function.

Example CloudTrail log

```
{
  "Records": [
    {
      "eventVersion": "1.02",
      "userIdentity": {
        "type": "Root",
        "principalId": "123456789012",
        "arn": "arn:aws:iam::123456789012:root",
        "accountId": "123456789012",
        "accessKeyId": "access-key-id",
        "sessionContext": {
          "attributes": {
            "mfaAuthenticated": "false",
            "creationDate": "2015-01-24T22:41:54Z"
          }
        }
      },
      "eventTime": "2015-01-24T23:26:50Z",
      "eventSource": "sns.amazonaws.com",
      "eventName": "CreateTopic",
      "awsRegion": "us-east-2",
      "sourceIPAddress": "205.251.233.176",
```

```
"userAgent":"console.amazonaws.com",
"requestParameters":{
    "name":"dropmeplease"
},
"responseElements":{
    "topicArn":"arn:aws:sns:us-east-2:123456789012:exampletopic"
},
"requestID":"3fdb7834-9079-557e-8ef2-350abc03536b",
"eventID":"17b46459-dada-4278-b8e2-5a4ca9ff1a9c",
"eventType":"AwsApiCall",
"recipientAccountId":"123456789012"
},
{
    "eventVersion":"1.02",
    "userIdentity":{
        "type":"Root",
        "principalId":"123456789012",
        "arn":"arn:aws:iam::123456789012:root",
        "accountId":"123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext":{
            "attributes":{
                "mfaAuthenticated":"false",
                "creationDate":"2015-01-24T22:41:54Z"
            }
        }
    },
    "eventTime":"2015-01-24T23:27:02Z",
    "eventSource":"sns.amazonaws.com",
    "eventName":"GetTopicAttributes",
    "awsRegion":"us-east-2",
    "sourceIPAddress":"205.251.233.176",
    "userAgent":"console.amazonaws.com",
    "requestParameters":{
        "topicArn":"arn:aws:sns:us-east-2:123456789012:exampletopic"
    },
    "responseElements":null,
    "requestID":"4a0388f7-a0af-5df9-9587-c5c98c29cbec",
    "eventID":"ec5bb073-8fa1-4d45-b03c-f07b9fc9ea18",
    "eventType":"AwsApiCall",
    "recipientAccountId":"123456789012"
}
]
```

For detailed information about how to configure Amazon S3 as the event source, see [Using AWS Lambda with Amazon S3 \(p. 194\)](#).

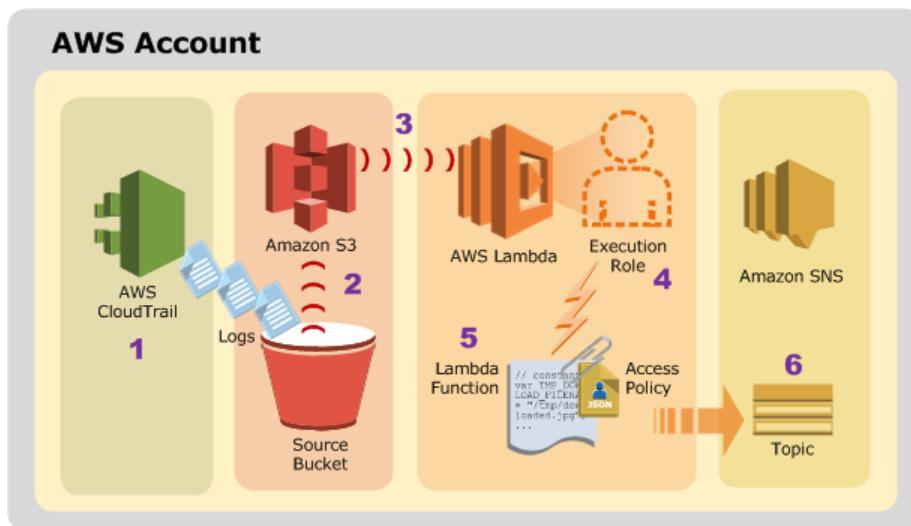
Topics

- [Tutorial: Using AWS Lambda with AWS CloudTrail \(p. 151\)](#)
- [Sample Function Code \(p. 157\)](#)

Tutorial: Using AWS Lambda with AWS CloudTrail

In this scenario, AWS CloudTrail will maintain records (logs) of AWS API calls made on your account and notify you anytime an API call is made to create an SNS topic. As API calls are made in your account, CloudTrail writes logs to an Amazon S3 bucket that you configured. In this scenario, you want Amazon S3 to publish the object-created events to AWS Lambda and invoke your Lambda function as CloudTrail creates log objects.

The following diagram summarizes the flow:



1. AWS CloudTrail saves logs to an S3 bucket (object-created event).
2. Amazon S3 detects the object-created event.
3. Amazon S3 publishes the `s3:ObjectCreated:*` event to AWS Lambda by invoking the Lambda function, as specified in the bucket notification configuration. Because the Lambda function's access permissions policy includes permissions for Amazon S3 to invoke the function, Amazon S3 can invoke the function.
4. AWS Lambda executes the Lambda function by assuming the execution role that you specified at the time you created the Lambda function.
5. The Lambda function reads the Amazon S3 event it receives as a parameter, determines where the CloudTrail object is, reads the CloudTrail object, and then it processes the log records in the CloudTrail object.
6. If the log includes a record with specific `eventType` and `eventSource` values, it publishes the event to your Amazon SNS topic. In [Tutorial: Using AWS Lambda with AWS CloudTrail \(p. 151\)](#), you subscribe to the SNS topic using the email protocol, so you get email notifications.

When Amazon S3 invokes your Lambda function, it passes an S3 event identifying, among other things, the bucket name and key name of the object that CloudTrail created. Your Lambda function can read the log object, and it knows the API calls that were reported in the log.

Each object CloudTrail creates in your S3 bucket is a JSON object, with one or more event records. Each record, among other things, provides `eventSource` and `eventName`.

```
{
  "Records": [
    {
      "eventVersion": "1.02",
      "userIdentity": {
        ...
      },
      "eventTime": "2014-12-16T19:17:43Z",
      "eventSource": "sns.amazonaws.com",
      "eventName": "CreateTopic",
      "awsRegion": "us-east-2",
      "sourceIPAddress": "72.21.198.64",
      ...
    },
    {
      ...
    }
  ]
}
```

```
    },
    ...
}
```

For illustration, the Lambda function notifies you by email if an API call to create an Amazon SNS topic is reported in the log. That is, when your Lambda function parses the log, it looks for records with the following:

- `eventSource = "sns.amazonaws.com"`
- `eventName = "CreateTopic"`

If found, it publishes the event to your Amazon SNS topic (you configure this topic to notify you by email).

Your Lambda function uses an S3 event that provides the bucket name and key name of the object CloudTrail created. Your Lambda function then reads that object to process CloudTrail records.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Getting Started with AWS Lambda \(p. 3\)](#) to create your first Lambda function.

To follow the procedures in this guide, you will need a command line terminal or shell to run commands. Commands are shown in listings preceded by a prompt symbol (\$) and the name of the current directory, when appropriate:

```
~/lambda-project$ this is a command
this is output
```

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

On Linux and macOS, use your preferred shell and package manager. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Turn on CloudTrail

In the AWS CloudTrail console, turn on the trail in your account by specifying a bucket for CloudTrail to save logs. When configuring the trail, do not enable SNS notifications.

For instructions, see [Creating and Updating Your Trail](#) in the *AWS CloudTrail User Guide*.

Create the Execution Role

Create the [execution role \(p. 9\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – AWS Lambda.**

- **Role name – lambda-cloudtrail-role.**
- **Permissions** – Custom policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "logs:*"  
            ],  
            "Resource": "arn:aws:logs:*:*:  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "s3:GetObject"  
            ],  
            "Resource": "arn:aws:s3:::my-bucket/*"  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "sns:Publish"  
            ],  
            "Resource": "arn:aws:sns:us-west-2:123456789012:my-topic"  
        }  
    ]  
}
```

The policy has the permissions that the function needs to read items from Amazon S3 and write logs to CloudWatch Logs.

Create the Function

The following example processes CloudTrail logs, and sends a notification when an Amazon SNS topic was created.

Example index.js

```
var aws = require('aws-sdk');  
var zlib = require('zlib');  
var async = require('async');  
  
var EVENT_SOURCE_TO_TRACK = /sns.amazonaws.com/;  
var EVENT_NAME_TO_TRACK = /CreateTopic/;  
var DEFAULT_SNS_REGION = 'us-east-2';  
var SNS_TOPIC_ARN = 'arn:aws:sns:us-west-2:123456789012:my-topic';  
  
var s3 = new aws.S3();  
var sns = new aws.SNS({  
    apiVersion: '2010-03-31',  
    region: DEFAULT_SNS_REGION  
});  
  
exports.handler = function(event, context, callback) {  
    var srcBucket = event.Records[0].s3.bucket.name;  
    var srcKey = event.Records[0].s3.object.key;  
  
    async.waterfall([  
        function fetchLogFromS3(next){  
            var params = {  
                Bucket: srcBucket,  
                Key: srcKey  
            };  
  
            s3.getObject(params, function(err, data){  
                if (err){  
                    return next(err);  
                }  
  
                var log = zlib.inflateSync(data.body);  
  
                var logEvent = JSON.parse(log);  
  
                if (logEvent.eventSource === EVENT_SOURCE_TO_TRACK && logEvent.eventType === EVENT_NAME_TO_TRACK){  
                    var snsParams = {  
                        TopicArn: SNS_TOPIC_ARN,  
                        Message: logEvent.message  
                    };  
  
                    sns.publish(snsParams, function(err, response){  
                        if (err){  
                            return next(err);  
                        }  
  
                        return next();  
                    });  
                }  
            });  
        }  
    ],  
    function done(err){  
        if (err){  
            return callback(err);  
        }  
  
        callback();  
    }  
};
```

```
        console.log('Fetching compressed log from S3...');
        s3.getObject({
            Bucket: srcBucket,
            Key: srcKey
        },
        next);
    },
    function uncompressLog(response, next){
        console.log("Uncompressing log...");
        zlib.gunzip(response.Body, next);
    },
    function publishNotifications(jsonBuffer, next) {
        console.log('Filtering log...');
        var json = jsonBuffer.toString();
        console.log('CloudTrail JSON from S3:', json);
        var records;
        try {
            records = JSON.parse(json);
        } catch (err) {
            next('Unable to parse CloudTrail JSON: ' + err);
            return;
        }
        var matchingRecords = records
            .Records
            .filter(function(record) {
                return record.eventSource.match(EVENT_SOURCE_TO_TRACK)
                    && record.eventName.match(EVENT_NAME_TO_TRACK);
            });

        console.log('Publishing ' + matchingRecords.length + ' notification(s) in
parallel...');
        async.each(
            matchingRecords,
            function(record, publishComplete) {
                console.log('Publishing notification: ', record);
                sns.publish({
                    Message:
                        'Alert... SNS topic created: \n TopicARN=' +
                    record.responseElements.topicArn + '\n\n' +
                        JSON.stringify(record),
                    TopicArn: SNS_TOPIC_ARN
                }, publishComplete);
            },
            next
        );
    },
    function (err) {
        if (err) {
            console.error('Failed to publish notifications: ', err);
        } else {
            console.log('Successfully published all notifications.');
        }
        callback(null,"message");
    });
};
```

To create the function

1. Copy the sample code into a file named `index.js` in a folder named `lambda-cloudtrail`.
2. Install `async` with `npm`.

```
~/lambda-cloudtrail$ npm install async
```

3. Create a deployment package.

```
~/lambda-cloudtrail$ zip -r function.zip .
```

4. Create a Lambda function with the `create-function` command.

```
$ aws lambda create-function --function-name CloudTrailEventProcessing \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs8.10 --timeout
10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/lambda-cloudtrail-role
```

Add Permissions to the Function Policy

Add permissions to the Lambda function's resource policy to allow Amazon S3 to invoke the function.

1. Run the following `add-permission` command to grant Amazon S3 service principal (`s3.amazonaws.com`) permissions to perform the `lambda:InvokeFunction` action. Note that permission is granted to Amazon S3 to invoke the function only if the following conditions are met:
 - An object-created event is detected on a specific bucket.
 - The bucket is owned by a specific AWS account. If a bucket owner deletes a bucket, some other AWS account can create a bucket with the same name. This condition ensures that only a specific AWS account can invoke your Lambda function.

```
$ aws lambda add-permission --function-name CloudTrailEventProcessing \
--statement-id Id-1 --action "lambda:InvokeFunction" --principal s3.amazonaws.com \
--source-arn arn:aws:s3:::my-bucket \
--source-account 123456789012
```

2. Verify the function's access policy with the `get-policy` command.

```
$ aws lambda get-policy --function-name function-name
```

Configure Notification on the Bucket

Add notification configuration on the bucket to request Amazon S3 to publish object-created events to Lambda. In the configuration, you specify the following:

- Event type – Any event types that create objects.
- Lambda function ARN – This is your Lambda function that you want Amazon S3 to invoke.

```
arn:aws:lambda:us-east-2:123456789012:function:CloudTrailEventProcessing
```

For instructions on adding notification configuration to a bucket, see [Enabling Event Notifications](#) in the *Amazon Simple Storage Service Console User Guide*.

Test the Setup

Now you can test the setup as follows:

1. Create an Amazon SNS topic.
2. AWS CloudTrail creates a log object in your bucket.

3. Amazon S3 invokes your Lambda function by passing in the log object's location as event data.
4. Lambda executes your function. The function retrieves the log, finds a CreateTopic event, and sends a notification.

Sample Function Code

Sample code is available for the following languages.

Topics

- [Node.js \(p. 157\)](#)

Node.js

The following example processes CloudTrail logs, and sends a notification when an Amazon SNS topic was created.

Example index.js

```
var aws = require('aws-sdk');
var zlib = require('zlib');
var async = require('async');

var EVENT_SOURCE_TO_TRACK = /sns.amazonaws.com/;
var EVENT_NAME_TO_TRACK = /CreateTopic/;
var DEFAULT_SNS_REGION = 'us-west-2';
var SNS_TOPIC_ARN = 'The ARN of your SNS topic';

var s3 = new aws.S3();
var sns = new aws.SNS({
    apiVersion: '2010-03-31',
    region: DEFAULT_SNS_REGION
});

exports.handler = function(event, context, callback) {
    var srcBucket = event.Records[0].s3.bucket.name;
    var srcKey = event.Records[0].s3.object.key;

    async.waterfall([
        function fetchLogFromS3(next){
            console.log('Fetching compressed log from S3...');
            s3.getObject({
                Bucket: srcBucket,
                Key: srcKey
            },
            next);
        },
        function uncompressLog(response, next){
            console.log("Uncompressing log...");
            zlib.gunzip(response.Body, next);
        },
        function publishNotifications(jsonBuffer, next) {
            console.log('Filtering log...');
            var json = jsonBuffer.toString();
            console.log('CloudTrail JSON from S3:', json);
            var records;
            try {
                records = JSON.parse(json);
            } catch (err) {
                next('Unable to parse CloudTrail JSON: ' + err);
                return;
            }
            var snsMessage = {
                TopicArn: SNS_TOPIC_ARN,
                Subject: 'CloudTrail Log',
                Message: 'A new CloudTrail log has been uploaded to ' + srcBucket + ' at ' + new Date().toString()
            };
            sns.publish(snsMessage, next);
        }
    ],
    function(err, result) {
        if (err) {
            callback(err);
        } else {
            callback(null, result);
        }
    }
});
```

```
        }
        var matchingRecords = records
            .Records
            .filter(function(record) {
                return record.eventSource.match(EVENT_SOURCE_TO_TRACK)
                    && record.eventName.match(EVENT_NAME_TO_TRACK);
            });

        console.log('Publishing ' + matchingRecords.length + ' notification(s) in
parallel...');
        async.each(
            matchingRecords,
            function(record, publishComplete) {
                console.log('Publishing notification: ', record);
                sns.publish({
                    Message:
                        'Alert... SNS topic created: \n TopicARN=' +
                    record.responseElements.topicArn + '\n\n' +
                        JSON.stringify(record),
                    TopicArn: SNS_TOPIC_ARN
                }, publishComplete);
            },
            next
        );
    }
], function (err) {
    if (err) {
        console.error('Failed to publish notifications: ', err);
    } else {
        console.log('Successfully published all notifications.');
    }
    callback(null,"message");
});
};

};
```

Zip up the sample code to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Node.js \(p. 241\)](#).

Using AWS Lambda with Amazon CloudWatch Events

Amazon CloudWatch Events help you to respond to state changes in your AWS resources. When your resources change state, they automatically send events into an event stream. You can create rules that match selected events in the stream and route them to your AWS Lambda function to take action. For example, you can automatically invoke an AWS Lambda function to log the state of an [EC2 instance](#) or [AutoScaling Group](#).

You maintain event source mapping in Amazon CloudWatch Events by using a rule target definition. For more information, see the [PutTargets](#) operation in the *Amazon CloudWatch Events API Reference*.

You can also create a Lambda function and direct AWS Lambda to execute it on a regular schedule. You can specify a fixed rate (for example, execute a Lambda function every hour or 15 minutes), or you can specify a Cron expression. For more information on expressions schedules, see [Schedule Expressions Using Rate or Cron \(p. 163\)](#).

Example CloudWatch Events Message Event

```
{
```

```
    "account": "123456789012",
    "region": "us-east-2",
    "detail": {},
    "detail-type": "Scheduled Event",
    "source": "aws.events",
    "time": "2019-03-01T01:23:45Z",
    "id": "cdc73f9d-aea9-11e3-9d5a-835b769c0d9c",
    "resources": [
        "arn:aws:events:us-east-1:123456789012:rule/my-schedule"
    ]
}
```

This functionality is available when you create a Lambda function using the AWS Lambda console or the AWS CLI. To configure it using the AWS CLI, see [Run an AWS Lambda Function on a Schedule Using the AWS CLI](#). The console provides **CloudWatch Events** as an event source. At the time of creating a Lambda function, you choose this event source and specify a time interval.

If you have made any manual changes to the permissions on your function, you may need to reapply the scheduled event access to your function. You can do that by using the following CLI command.

```
$ aws lambda add-permission --function-name my-function
  --action 'lambda:InvokeFunction' --principal events.amazonaws.com --statement-id
  events-access \
  --source-arn arn:aws:events:*:123456789012:rule/*
```

Each AWS account can have up to 100 unique event sources of the **CloudWatch Events- Schedule** source type. Each of these can be the event source for up to five Lambda functions. That is, you can have up to 500 Lambda functions that can be executing on a schedule in your AWS account.

The console also provides a blueprint (**lambda-canary**) that uses the **CloudWatch Events - Schedule** source type. Using this blueprint, you can create a sample Lambda function and test this feature. The example code that the blueprint provides checks for the presence of a specific webpage and specific text string on the webpage. If either the webpage or the text string is not found, the Lambda function throws an error.

Tutorial: Using AWS Lambda with Scheduled Events

In this tutorial, you do the following:

- Create a Lambda function using the **lambda-canary** blueprint. You configure the Lambda function to run every minute. Note that if the function returns an error, AWS Lambda logs error metrics to CloudWatch.
- Configure a CloudWatch alarm on the `Errors` metric of your Lambda function to post a message to your Amazon SNS topic when AWS Lambda emits error metrics to CloudWatch. You subscribe to the Amazon SNS topics to get email notification. In this tutorial, you do the following to set this up:
 - Create an Amazon SNS topic.
 - Subscribe to the topic so you can get email notifications when a new message is posted to the topic.
 - In Amazon CloudWatch, set an alarm on the `Errors` metric of your Lambda function to publish a message to your SNS topic when errors occur.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Getting Started with AWS Lambda \(p. 3\)](#) to create your first Lambda function.

Create a Lambda Function

1. Sign in to the AWS Management Console and open the AWS Lambda console at <https://console.aws.amazon.com/lambda/>.
2. Choose **Create function**.
3. Choose **Blueprints**.
4. Enter **canary** in the search bar. Choose the **lambda-canary** blueprint and then choose **Configure**.
5. Configure the following settings.
 - **Name** – **lambda-canary**.
 - **Role** – **Create a new role from one or more templates**.
 - **Role name** – **lambda-apigateway-role**.
 - **Policy templates** – **Simple microservice permissions**.
 - **Rule** – **Create a new rule**.
 - **Rule name** – **CheckWebsiteScheduledEvent**.
 - **Rule description** – **CheckWebsiteScheduledEvent trigger**.
 - **Schedule expression** – **rate(1 minute)**.
 - **Enabled** – True (checked).
 - **Environment variables**
 - **site** – <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>
 - **expected** – **What Is AWS Lambda?**
6. Choose **Create function**.

CloudWatch Events emits an event every minute, based on the schedule expression. The event triggers the Lambda function, which verifies that the expected string appears in the specified page. For more information on expressions schedules, see [Schedule Expressions Using Rate or Cron \(p. 163\)](#).

Test the Lambda Function

Test the function with a sample event provided by the Lambda console.

1. Open the Lambda console [Functions page](#).
2. Choose **lambda-canary**.
3. Next to the **Test** button at the top of the page, choose **Configure test events** from the drop-down menu.
4. Create a new event using the **CloudWatch Events** event template.
5. Choose **Create**.
6. Choose **Test**.

The output from the function execution is shown at the top of the page.

Create an Amazon SNS Topic and Subscribe to It

Create an Amazon Simple Notification Service topic to receive notifications when the canary function returns an error.

To create a topic

1. Open the [Amazon SNS console](#).

2. Choose **Create topic**.
3. Create a topic with the following settings.
 - **Name** – `lambda-canary-notifications`.
 - **Display name** – `Canary`.
4. Choose **Create subscription**.
5. Create a subscription with the following settings.
 - **Protocol** – `Email`.
 - **Endpoint** – Your email address.

Amazon SNS sends an email from `Canary <no-reply@sns.amazonaws.com>`, reflecting the friendly name of the topic. Use the link in the email to confirm your address.

Configure an Alarm

Configure an alarm in Amazon CloudWatch that monitors the Lambda function and sends a notification when it fails.

To create an alarm

1. Open the [CloudWatch console](#).
2. Choose **Alarms**.
3. Choose **Create alarm**.
4. Choose **Alarms**.
5. Create an alarm with the following settings.
 - **Metrics** – `lambda-canary Errors`.
Search for `lambda canary errors` to find the metric.
 - **Statistic** – `Sum`.
Choose the statistic from the drop down above the preview graph.
 - **Name** – `lambda-canary-alarm`.
 - **Description** – `Lambda canary alarm`.
 - **Threshold** – `Whenever Errors is >=1`.
 - **Send notification to** – `lambda-canary-notifications`

Test the Alarm

Update the function configuration to cause the function to return an error, triggering the alarm.

To trigger an alarm

1. Open the Lambda console [Functions page](#).
2. Choose `lambda-canary`.
3. Under **Environment variables**, set **expected** to `404`.
4. Choose **Save**

Wait a minute, and then check your email for a message from Amazon SNS

AWS SAM Template for a CloudWatch Events Application

You can build this application using [AWS SAM](#). To learn more about creating AWS SAM templates, see [AWS SAM Template Basics](#) in the *AWS Serverless Application Model Developer Guide*.

Below is a sample AWS SAM template for the Lambda application from the [tutorial \(p. 159\)](#). Copy the text below to a `.yaml` file and save it next to the ZIP package you created previously. Note that the Handler and Runtime parameter values should match the ones you used when you created the function in the previous section.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Parameters:
  NotificationEmail:
    Type: String
Resources:
  CheckWebsitePeriodically:
    Type: AWS::Serverless::Function
    Properties:
      Handler: LambdaFunctionOverHttps.handler
      Runtime: runtime
      Policies: AmazonDynamoDBFullAccess
    Events:
      CheckWebsiteScheduledEvent:
        Type: Schedule
        Properties:
          Schedule: rate(1 minute)

  AlarmTopic:
    Type: AWS::SNS::Topic
    Properties:
      Subscription:
        - Protocol: email
          Endpoint: !Ref NotificationEmail

  Alarm:
    Type: AWS::CloudWatch::Alarm
    Properties:
      AlarmActions:
        - !Ref AlarmTopic
      ComparisonOperator: GreaterThanOrEqualToThreshold
      Dimensions:
        - Name: FunctionName
          Value: !Ref CheckWebsitePeriodically
      EvaluationPeriods: 1
      MetricName: Errors
      Namespace: AWS/Lambda
      Period: 60
      Statistic: Sum
      Threshold: '1'
```

For information on how to package and deploy your serverless application using the package and deploy commands, see [Deploying Serverless Applications](#) in the *AWS Serverless Application Model Developer Guide*.

Schedule Expressions Using Rate or Cron

AWS Lambda supports standard rate and cron expressions for frequencies of up to once per minute. CloudWatch Events rate expressions have the following format.

```
rate(Value Unit)
```

Where **Value** is a positive integer and **Unit** can be minute(s), hour(s), or day(s). For a singular value the unit must be singular (for example, `rate(1 day)`), otherwise plural (for example, `rate(5 days)`).

Rate Expression Examples

Frequency	Expression
Every 5 minutes	<code>rate(5 minutes)</code>
Every hour	<code>rate(1 hour)</code>
Every seven days	<code>rate(7 days)</code>

Cron expressions have the following format.

```
cron(Minutes Hours Day-of-month Month Day-of-week Year)
```

Cron Expression Examples

Frequency	Expression
10:15 AM (UTC) every day	<code>cron(15 10 * * ? *)</code>
6:00 PM Monday through Friday	<code>cron(0 18 ? * MON-FRI *)</code>
8:00 AM on the first day of the month	<code>cron(0 8 1 * ? *)</code>
Every 10 min on weekdays	<code>cron(0/10 * ? * MON-FRI *)</code>
Every 5 minutes between 8:00 AM and 5:55 PM weekdays	<code>cron(0/5 8-17 ? * MON-FRI *)</code>
9:00 AM on the first Monday of each month	<code>cron(0 9 ? * 2#1 *)</code>

Note the following:

- If you are using the Lambda console, do not include the `cron` prefix to your expression.
- One of the day-of-month or day-of-week values must be a question mark (?).

For more information, see [Schedule Expressions for Rules](#) in the *CloudWatch Events User Guide*.

Using AWS Lambda with Amazon CloudWatch Logs

You can use a Lambda function to monitor and analyze logs from an Amazon CloudWatch Logs log stream. Create [subscriptions](#) for one or more log streams to invoke a function when logs are created or match an optional pattern. Use the function to send a notification or persist the log to a database or storage.

CloudWatch Logs invokes your function asynchronously with an event that contains encoded log data.

Example Amazon CloudWatch Logs Message Event

```
{  
  "awslogs": {  
    "data":  
      "ewogICAgIm1lc3NhZ2VUeXB1IjogIkRBVEFFTUVTUOFHRSIsCiAgICAib3duZXIIoAiMTIzNDU2Nzg5MDEyIiwKICAgICJsb2dH  
  }  
}
```

When decoded, the log data is a JSON document with the following structure.

Example Amazon CloudWatch Logs Message Data (decoded)

```
{  
  "messageType": "DATA_MESSAGE",  
  "owner": "123456789012",  
  "logGroup": "/aws/lambda/echo-nodejs",  
  "logStream": "2019/03/13/[$LATEST]94fa867e5374431291a7fc14e2f56ae7",  
  "subscriptionFilters": [  
    "LambdaStream_cloudwatchlogs-node"  
  ],  
  "logEvents": [  
    {  
      "id": "34622316099697884706540976068822859012661220141643892546",  
      "timestamp": 1552518348220,  
      "message": "REPORT RequestId: 6234bffe-149a-b642-81ff-2e8e376d8aff\\tDuration:  
46.84 ms\\tBilled Duration: 100 ms \\tMemory Size: 192 MB\\tMax Memory Used: 72 MB\\t\\n"  
    }  
  ]  
}
```

For a sample application that uses CloudWatch Logs as a trigger, see [Error Processor Sample Application for AWS Lambda \(p. 119\)](#).

Using AWS Lambda with AWS CloudFormation

In an AWS CloudFormation template, you can specify a Lambda function as the target of a custom resource. Use custom resources to process parameters, retrieve configuration values, or call other AWS services during stack lifecycle events.

The following example invokes a function that's defined elsewhere in the template.

Example – Custom Resource Definition

```
Resources:  
  primerinvoke:
```

```
Type: AWS::CloudFormation::CustomResource
Version: "1.0"
Properties:
  ServiceToken: !GetAtt primer.Arn
  FunctionName: !Ref randomerror
```

The service token is the Amazon Resource Name (ARN) of the function that AWS CloudFormation invokes when you create, update, or delete the stack. You can also include additional properties like `FunctionName`, which AWS CloudFormation passes to your function as is.

AWS CloudFormation invokes your Lambda function asynchronously with an event that includes a callback URL.

Example – AWS CloudFormation Message Event

```
{
  "RequestType": "Create",
  "ServiceToken": "arn:aws:lambda:us-east-2:123456789012:function:lambda-error-processor-primer-14ROR2T3JKU66",
  "ResponseURL": "https://cloudformation-custom-resource-response-useast2.s3-us-east-2.amazonaws.com/arn%3Aaws%3Acloudformation%3Aus-east-2%3A123456789012%3Astack/lambda-error-processor/1134083a-2608-1e91-9897-022501a2c456%7Cprimerinvoke%7C5d478078-13e9-baf0-464a-7ef285ecc786?
AWSAccessKeyId=AKIAIOSFODNN7EXAMPLE&Expires=1555451971&Signature=28UiJZePE5I4dvukKQqM%2F9Rf1o4%3D",
  "StackId": "arn:aws:cloudformation:us-east-2:123456789012:stack/lambda-error-processor/1134083a-2608-1e91-9897-022501a2c456",
  "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
  "LogicalResourceId": "primerinvoke",
  "ResourceType": "AWS::CloudFormation::CustomResource",
  "ResourceProperties": {
    "ServiceToken": "arn:aws:lambda:us-east-2:123456789012:function:lambda-error-processor-primer-14ROR2T3JKU66",
    "FunctionName": "lambda-error-processor-randomerror-ZWUC391MQAJK"
  }
}
```

The function is responsible for returning a response to the callback URL that indicates success or failure. For the full response syntax, see [Custom Resource Response Objects](#).

Example – AWS CloudFormation Custom Resource Response

```
{
  "Status": "SUCCESS",
  "PhysicalResourceId": "2019/04/18/[${LATEST}]b3d1bfc65f19ec610654e4d9b9de47a0",
  "StackId": "arn:aws:cloudformation:us-east-2:123456789012:stack/lambda-error-processor/1134083a-2608-1e91-9897-022501a2c456",
  "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
  "LogicalResourceId": "primerinvoke"
}
```

AWS CloudFormation provides a library called `cfn-response` that handles sending the response. If you define your function within a template, you can require the library by name. AWS CloudFormation then adds the library to the deployment package that it creates for the function.

The following example function invokes a second function. If the call succeeds, the function sends a success response to AWS CloudFormation, and the stack update continues.

Example – Custom Resource Function

```
Resources:
```

```

primer:
  Type: AWS::Serverless::Function
  Properties:
    Handler: index.handler
    Runtime: nodejs8.10
    InlineCode: |
      var aws = require('aws-sdk');
      var response = require('cfn-response');
      exports.handler = function(event, context) {
        // For Delete requests, immediately send a SUCCESS response.
        if (event.RequestType == "Delete") {
          response.send(event, context, "SUCCESS");
          return;
        }
        var responseStatus = "FAILED";
        var responseData = {};
        var functionName = event.ResourceProperties.FunctionName
        var lambda = new aws.Lambda();
        lambda.invoke({ FunctionName: functionName }, function(err, invokeResult) {
          if (err) {
            responseData = {Error: "Invoke call failed"};
            console.log(responseData.Error + ":\n", err);
          }
          else responseStatus = "SUCCESS";
          response.send(event, context, responseStatus, responseData);
        });
      };
      Description: Invoke a function to create a log stream.
      MemorySize: 128
      Timeout: 8
      Role: !GetAtt role.Arn
      Tracing: Active

```

If the function that the custom resource invokes isn't defined in a template, you can get the source code for `cfn-response` from [cfn-response Module](#) in the AWS CloudFormation User Guide.

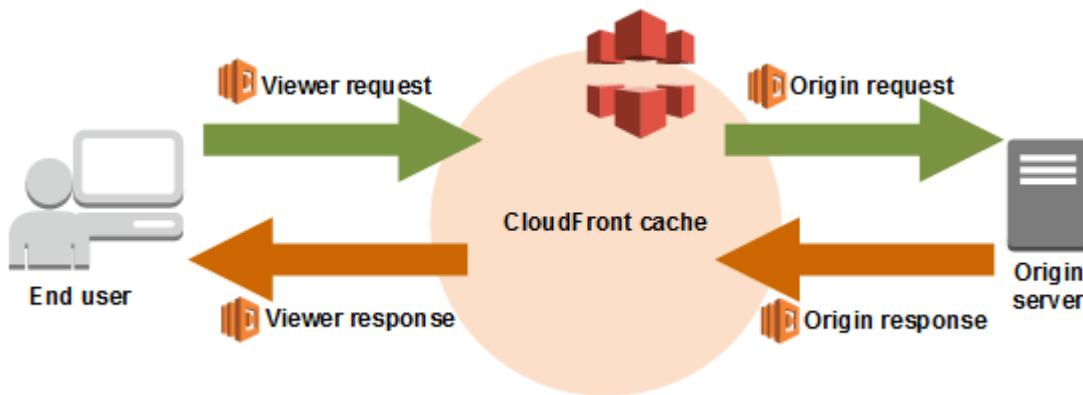
For a sample application that uses a custom resource to ensure that a function's log group is created before another resource that depends on it, see [Error Processor Sample Application for AWS Lambda \(p. 119\)](#).

For more information about custom resources, see [Custom Resources](#) in the *AWS CloudFormation User Guide*.

Using AWS Lambda with CloudFront Lambda@Edge

Lambda@Edge lets you run Node.js Lambda functions to customize content that CloudFront delivers, executing the functions in AWS locations closer to the viewer. The functions run in response to CloudFront events, without provisioning or managing servers. You can use Lambda functions to change CloudFront requests and responses at the following points:

- After CloudFront receives a request from a viewer (viewer request)
- Before CloudFront forwards the request to the origin (origin request)
- After CloudFront receives the response from the origin (origin response)
- Before CloudFront forwards the response to the viewer (viewer response)



You can also generate responses to viewers without ever sending the request to the origin.

Example CloudFront Message Event

```
{
  "Records": [
    {
      "cf": {
        "config": {
          "distributionId": "EDFDVBD6EXAMPLE"
        },
        "request": {
          "clientIp": "2001:0db8:85a3:0:0:8a2e:0370:7334",
          "method": "GET",
          "uri": "/picture.jpg",
          "headers": {
            "host": [
              {
                "key": "Host",
                "value": "d111111abcdef8.cloudfront.net"
              }
            ],
            "user-agent": [
              {
                "key": "User-Agent",
                "value": "curl/7.51.0"
              }
            ]
          }
        }
      }
    ]
  }
}
```

With Lambda@Edge, you can build a variety of solutions, for example:

- Inspect cookies to rewrite URLs to different versions of a site for A/B testing.
- Send different objects to your users based on the `User-Agent` header, which contains information about the device that submitted the request. For example, you can send images in different resolutions to users based on their devices.
- Inspect headers or authorized tokens, inserting a corresponding header and allowing access control before forwarding a request to the origin.
- Add, delete, and modify headers, and rewrite the URL path to direct users to different objects in the cache.

- Generate new HTTP responses to do things like redirect unauthenticated users to login pages, or create and deliver static webpages right from the edge. For more information, see [Using Lambda Functions to Generate HTTP Responses to Viewer and Origin Requests](#) in the *Amazon CloudFront Developer Guide*.

For more information about using Lambda@Edge, see [Using CloudFront with Lambda@Edge](#).

Using AWS Lambda with AWS CodeCommit

You can create a trigger for an AWS CodeCommit repository so that events in the repository will invoke a Lambda function. For example, you can invoke a Lambda function when a branch or tag is created or when a push is made to an existing branch.

Example AWS CodeCommit Message Event

```
{  
    "Records": [  
        {  
            "awsRegion": "us-east-2",  
            "codecommit": {  
                "references": [  
                    {  
                        "commit": "5e493c6f3067653f3d04eca608b4901eb227078",  
                        "ref": "refs/heads/master"  
                    }  
                ]  
            },  
            "eventId": "31ade2c7-f889-47c5-a937-1cf99e2790e9",  
            "eventName": "ReferenceChanges",  
            "eventPartNumber": 1,  
            "eventSource": "aws:codecommit",  
            "eventSourceARN": "arn:aws:codecommit:us-east-2:123456789012:lambda-pipeline-repo",  
            "eventTime": "2019-03-12T20:58:25.400+0000",  
            "eventTotalParts": 1,  
            "eventTriggerConfigId": "0d17d6a4-efeb-46f3-b3ab-a63741badeb8",  
            "eventTriggerName": "index.handler",  
            "eventVersion": "1.0",  
            "userIdentityARN": "arn:aws:iam::123456789012:user/intern"  
        }  
    ]  
}
```

For more information, see [Manage Triggers for an AWS CodeCommit Repository](#).

Using AWS Lambda with Amazon Cognito

The Amazon Cognito Events feature enables you to run Lambda functions in response to events in Amazon Cognito. For example, you can invoke a Lambda function for the Sync Trigger events, that is published each time a dataset is synchronized. To learn more and walk through an example, see [Introducing Amazon Cognito Events: Sync Triggers](#) in the Mobile Development blog.

Example Amazon Cognito Message Event

```
{
```

```

    "datasetName": "datasetName",
    "eventType": "SyncTrigger",
    "region": "us-east-1",
    "identityId": "identityId",
    "datasetRecords": {
        "SampleKey2": {
            "newValue": "newValue2",
            "oldValue": "oldValue2",
            "op": "replace"
        },
        "SampleKey1": {
            "newValue": "newValue1",
            "oldValue": "oldValue1",
            "op": "replace"
        }
    },
    "identityPoolId": "identityPoolId",
    "version": 2
}

```

You configure event source mapping using Amazon Cognito event subscription configuration. For information about event source mapping and a sample event, see [Amazon Cognito Events](#) in the *Amazon Cognito Developer Guide*.

Using AWS Lambda with AWS Config

You can use AWS Lambda functions to evaluate whether your AWS resource configurations comply with your custom Config rules. As resources are created, deleted, or changed, AWS Config records these changes and sends the information to your Lambda functions. Your Lambda functions then evaluate the changes and report results to AWS Config. You can then use AWS Config to assess overall resource compliance: you can learn which resources are noncompliant and which configuration attributes are the cause of noncompliance.

Example AWS Config Message Event

```

{
    "invokingEvent": "{\"configurationItem\":{\"configurationItemCaptureTime\":
\"2016-02-17T01:36:34.043Z\", \"awsAccountId\":\"000000000000\", \"configurationItemStatus\":
\"OK\", \"resourceId\":\"i-00000000\", \"ARN\":\"arn:aws:ec2:us-east-1:000000000000:instance/
i-00000000\", \"awsRegion\":\"us-east-1\", \"availabilityZone\":\"us-east-1a\",
\"resourceType\":\"AWS::EC2::Instance\", \"tags\":[{\"Foo\":\"Bar\"}], \"relationships\":
[{\\"resourceId\":\"eipalloc-00000000\", \"resourceType\":\"AWS::EC2::EIP\", \"name\":
\"Is attached to Elasticip\"}], \"configuration\":[{\"foo\":\"bar\"}], \"messageType\":
\"ConfigurationItemChangeNotification\"},
    \"ruleParameters\": {\"myParameterKey\":\"myParameterValue\"},
    \"resultToken\": \"myResultToken\",
    \"eventLeftScope\": false,
    \"executionRoleArn\": \"arn:aws:iam::012345678912:role/config-role\",
    \"configRuleArn\": \"arn:aws:config:us-east-1:012345678912:config-rule/config-
rule-0123456\",
    \"configRuleName\": \"change-triggered-config-rule\",
    \"configRuleId\": \"config-rule-0123456\",
    \"accountId\": \"012345678912\",
    \"version\": \"1.0\"}
}

```

For more information, see [Evaluating Resources With AWS Config Rules](#).

Using AWS Lambda with Amazon DynamoDB

You can use a AWS Lambda function to process records in a [Amazon DynamoDB Streams stream](#). With DynamoDB Streams, you can trigger a Lambda function to perform additional work each time a DynamoDB table is updated.

Lambda reads records from the stream and invokes your function [synchronously \(p. 82\)](#) with an event that contains stream records. Lambda reads records in batches and invokes your function to process records from the batch.

Example DynamoDB Streams Record Event

```
{  
    "Records": [  
        {  
            "eventID": "1",  
            "eventVersion": "1.0",  
            "dynamodb": {  
                "Keys": {  
                    "Id": {  
                        "N": "101"  
                    }  
                },  
                "NewImage": {  
                    "Message": {  
                        "S": "New item!"  
                    },  
                    "Id": {  
                        "N": "101"  
                    }  
                },  
                "StreamViewType": "NEW_AND_OLD_IMAGES",  
                "SequenceNumber": "111",  
                "SizeBytes": 26  
            },  
            "awsRegion": "us-west-2",  
            "eventName": "INSERT",  
            "eventSourceARN": "eventsourcearn",  
            "eventSource": "aws:dynamodb"  
        },  
        {  
            "eventID": "2",  
            "eventVersion": "1.0",  
            "dynamodb": {  
                "OldImage": {  
                    "Message": {  
                        "S": "New item!"  
                    },  
                    "Id": {  
                        "N": "101"  
                    }  
                },  
                "SequenceNumber": "222",  
                "Keys": {  
                    "Id": {  
                        "N": "101"  
                    }  
                },  
                "SizeBytes": 59,  
                "NewImage": {  
                    "Message": {  
                        "S": "This item has changed"  
                    },  
                    "Id": {  
                        "N": "101"  
                    }  
                }  
            }  
        }  
    ]  
}
```

```
        "Id": {  
            "N": "101"  
        },  
        "StreamViewType": "NEW_AND_OLD_IMAGES"  
    },  
    "awsRegion": "us-west-2",  
    "eventName": "MODIFY",  
    "eventSourceARN": sourcearn,  
    "eventSource": "aws:dynamodb"  
}
```

Lambda polls shards in your DynamoDB Streams stream for records at a base rate of 4 times per second. When records are available, Lambda invokes your function and waits for the result. If processing succeeds, Lambda resumes polling until it receives more records.

If your function returns an error, Lambda retries the batch until processing succeeds or the data expires. Until the issue is resolved, no data in the shard is processed. Handle any record processing errors in your code to avoid stalled shards and potential data loss.

Topics

- [Creating an Event Source Mapping \(p. 171\)](#)
- [Execution Role Permissions \(p. 172\)](#)
- [Tutorial: Using AWS Lambda with Amazon DynamoDB Streams \(p. 172\)](#)
- [Sample Function Code \(p. 176\)](#)
- [AWS SAM Template for a DynamoDB Application \(p. 179\)](#)

Creating an Event Source Mapping

Create an event source mapping to tell Lambda to send records from your stream to a Lambda function. You can create multiple event source mappings to process the same data with multiple Lambda functions, or process items from multiple streams with a single function.

To configure your function to read from DynamoDB Streams in the Lambda console, create a **DynamoDB** trigger.

To create a trigger

1. Open the Lambda console [Functions page](#).
2. Choose a function.
3. Under **Designer**, choose a trigger type to add a trigger to your function.
4. Under **Configure triggers**, configure the required options and then choose **Add**.
5. Choose **Save**.

Event Source Options

- **DynamoDB table** – The DynamoDB table from which to read records.
- **Batch size** – The number of records to read from a shard in each batch, up to 1,000. Lambda passes all of the records in the batch to the function in a single call, as long as the total size of the events doesn't exceed the payload limit of 6 MB.
- **Starting position** – Process only new records, or all existing records.
 - **Latest** – Process new records added to the stream.
 - **Trim horizon** – Process all records in the stream.

After processing any existing records, the function is caught up and continues to process new records.

- **Enabled** – Disable the event source to stop processing records. Lambda keeps track of the last record processed and resumes processing from that point when re-enabled.

To manage the event source configuration later, choose the trigger in the designer.

Execution Role Permissions

Lambda needs the following permissions to manage resources related to your DynamoDB Streams stream. Add them to your function's [execution role \(p. 9\)](#).

- [dynamodb:DescribeStream](#)
- [dynamodb:GetRecords](#)
- [dynamodb:GetShardIterator](#)
- [dynamodb>ListStreams](#)

The `AWSLambdaDynamoDBExecutionRole` managed policy includes these permissions. For more information, see [AWS Lambda Execution Role \(p. 9\)](#).

Tutorial: Using AWS Lambda with Amazon DynamoDB Streams

In this tutorial, you create a Lambda function to consume events from an Amazon DynamoDB stream.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Getting Started with AWS Lambda \(p. 3\)](#) to create your first Lambda function.

To follow the procedures in this guide, you will need a command line terminal or shell to run commands. Commands are shown in listings preceded by a prompt symbol (\$) and the name of the current directory, when appropriate:

```
~/lambda-project$ this is a command  
this is output
```

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

On Linux and macOS, use your preferred shell and package manager. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the Execution Role

Create the [execution role \(p. 9\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.

- **Trusted entity** – Lambda.
- **Permissions** – **AWSLambdaDynamoDBExecutionRole**.
- **Role name** – **lambda-dynamodb-role**.

The **AWSLambdaDynamoDBExecutionRole** has the permissions that the function needs to read items from DynamoDB and write logs to CloudWatch Logs.

Create the Function

The following example code receives a DynamoDB event input and processes the messages that it contains. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Note

For sample code in other languages, see [Sample Function Code \(p. 176\)](#).

Example index.js

```
console.log('Loading function');

exports.handler = function(event, context, callback) {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        console.log(record.eventID);
        console.log(record.eventName);
        console.log('DynamoDB Record: %j', record.dynamodb);
    });
    callback(null, "message");
};
```

To create the function

1. Copy the sample code into a file named `index.js`.
2. Create a deployment package.

```
$ zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command.

```
$ aws lambda create-function --function-name ProcessDynamoDBRecords \
--zip-file file://function.zip --handler index.handler --runtime nodejs8.10 \
--role arn:aws:iam::123456789012:role/lambda-dynamodb-role
```

Test the Lambda Function

In this step, you invoke your Lambda function manually using the `invoke` AWS CLI command and the following sample DynamoDB event.

Example input.txt

```
{
  "Records": [
    {
      "eventID": "1",
      "eventName": "INSERT",
```

```

    "eventVersion":"1.0",
    "eventSource":"aws:dynamodb",
    "awsRegion":"us-east-1",
    "dynamodb":{
        "Keys":{},
        "Id":{
            "N":"101"
        }
    },
    "NewImage":{
        "Message":{
            "S":"New item!"
        },
        "Id":{
            "N":"101"
        }
    },
    "SequenceNumber":"111",
    "SizeBytes":26,
    "StreamViewType":"NEW_AND_OLD_IMAGES"
},
"eventSourceARN":"stream-ARN"
},
{
    "eventID":"2",
    "eventName":"MODIFY",
    "eventVersion":"1.0",
    "eventSource":"aws:dynamodb",
    "awsRegion":"us-east-1",
    "dynamodb":{
        "Keys":{},
        "Id":{
            "N":"101"
        }
    },
    "NewImage":{
        "Message":{
            "S":"This item has changed"
        },
        "Id":{
            "N":"101"
        }
    },
    "OldImage":{
        "Message":{
            "S":"New item!"
        },
        "Id":{
            "N":"101"
        }
    },
    "SequenceNumber":"222",
    "SizeBytes":59,
    "StreamViewType":"NEW_AND_OLD_IMAGES"
},
"eventSourceARN":"stream-ARN"
},
{
    "eventID":"3",
    "eventName":"REMOVE",
    "eventVersion":"1.0",
    "eventSource":"aws:dynamodb",
    "awsRegion":"us-east-1",
    "dynamodb":{
        "Keys":{},
        "Id":{


```

```
        "N":"101"
    }
},
"OldImage":{
    "Message":{
        "S":"This item has changed"
    },
    "Id":{
        "N":"101"
    }
},
"SequenceNumber":"333",
"SizeBytes":38,
"StreamViewType":"NEW_AND_OLD_IMAGES"
},
"eventSourceARN":"stream-ARN"
}
]
```

Execute the following `invoke` command.

```
$ aws lambda invoke --function-name ProcessDynamoDBRecords --payload file://input.txt
outputfile.txt
```

The function returns the string message (message in the `context.succeed()` in the code) in the response body.

Verify the output in the `outputfile.txt` file.

Create a DynamoDB Table with a Stream Enabled

Create an Amazon DynamoDB table with a stream enabled.

To create a DynamoDB table

1. Open the [DynamoDB console](#).
2. Choose **Create table**.
3. Create a table with the following settings.
 - **Table name** – `lambda-dynamodb-stream`
 - **Primary key** – `id` (string)
4. Choose **Create**.

To enable streams

1. Open the [DynamoDB console](#).
2. Choose **Tables**.
3. Choose the `lambda-dynamodb-stream` table.
4. Under **Overview**, choose **Manage stream**.
5. Choose **Enable**.

Write down the stream ARN. You need this in the next step when you associate the stream with your Lambda function. For more information on enabling streams, see [Capturing Table Activity with DynamoDB Streams](#).

Add an Event Source in AWS Lambda

Create an event source mapping in AWS Lambda. This event source mapping associates the DynamoDB stream with your Lambda function. After you create this event source mapping, AWS Lambda starts polling the stream.

Run the following AWS CLI `create-event-source-mapping` command. After the command executes, note down the UUID. You'll need this UUID to refer to the event source mapping in any commands, for example, when deleting the event source mapping.

```
$ aws lambda create-event-source-mapping --function-name ProcessDynamoDBRecords \
--batch-size 100 --starting-position LATEST --event-source DynamoDB-stream-arn
```

This creates a mapping between the specified DynamoDB stream and the Lambda function. You can associate a DynamoDB stream with multiple Lambda functions, and associate the same Lambda function with multiple streams. However, the Lambda functions will share the read throughput for the stream they share.

You can get the list of event source mappings by running the following command.

```
$ aws lambda list-event-source-mappings
```

The list returns all of the event source mappings you created, and for each mapping it shows the `LastProcessingResult`, among other things. This field is used to provide an informative message if there are any problems. Values such as `No records processed` (indicates that AWS Lambda has not started polling or that there are no records in the stream) and `OK` (indicates AWS Lambda successfully read records from the stream and invoked your Lambda function) indicate that there are no issues. If there are issues, you receive an error message.

If you have a lot of event source mappings, use the `function-name` parameter to narrow down the results.

```
$ aws lambda list-event-source-mappings --function-name ProcessDynamoDBRecords
```

Test the Setup

Test the end-to-end experience. As you perform table updates, DynamoDB writes event records to the stream. As AWS Lambda polls the stream, it detects new records in the stream and executes your Lambda function on your behalf by passing events to the function.

1. In the DynamoDB console, add, update, delete items to the table. DynamoDB writes records of these actions to the stream.
2. AWS Lambda polls the stream and when it detects updates to the stream, it invokes your Lambda function by passing in the event data it finds in the stream.
3. Your function executes and creates logs in Amazon CloudWatch. You can verify the logs reported in the Amazon CloudWatch console.

Sample Function Code

Sample code is available for the following languages.

Topics

- [Node.js \(p. 177\)](#)

- [Java 8 \(p. 177\)](#)
- [C# \(p. 178\)](#)
- [Python 3 \(p. 179\)](#)
- [Go \(p. 179\)](#)

Node.js

The following example processes messages from DynamoDB, and logs their contents.

Example ProcessDynamoDBStream.js

```
console.log('Loading function');

exports.lambda_handler = function(event, context, callback) {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        console.log(record.eventID);
        console.log(record.eventName);
        console.log('DynamoDB Record: %j', record.dynamodb);
    });
    callback(null, "message");
};
```

Zip up the sample code to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Node.js \(p. 241\)](#).

Java 8

The following example processes messages from DynamoDB, and logs their contents. `handleRequest` is the handler that AWS Lambda invokes and provides event data. The handler uses the predefined `DynamodbEvent` class, which is defined in the `aws-lambda-java-events` library.

Example DDBEventProcessor.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;

public class DDBEventProcessor implements
    RequestHandler<DynamodbEvent, String> {

    public String handleRequest(DynamodbEvent ddbEvent, Context context) {
        for (DynamodbStreamRecord record : ddbEvent.getRecords()){
            System.out.println(record.getEventID());
            System.out.println(record.geteventName());
            System.out.println(record.getDynamodb().toString());

        }
        return "Successfully processed " + ddbEvent.getRecords().size() + " records.";
    }
}
```

If the handler returns normally without exceptions, Lambda considers the input batch of records as processed successfully and begins reading new records in the stream. If the handler throws an exception,

Lambda considers the input batch of records as not processed and invokes the function with the same batch of records again.

Dependencies

- aws-lambda-java-core
- aws-lambda-java-events

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Java \(p. 264\)](#).

C#

The following example processes messages from DynamoDB, and logs their contents.

`ProcessDynamoEvent` is the handler that AWS Lambda invokes and provides event data. The handler uses the predefined `DynamoDbEvent` class, which is defined in the `Amazon.Lambda.DynamoDBEvents` library.

Example ProcessingDynamoDBStreams.cs

```
using System;
using System.IO;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

using Amazon.Lambda.Serialization.Json;

namespace DynamoDBStreams
{
    public class DdbSample
    {
        private static readonly JsonSerializer _jsonSerializer = new JsonSerializer();

        public void ProcessDynamoEvent(DynamoDBEvent dynamoEvent)
        {
            Console.WriteLine($"Beginning to process {dynamoEvent.Records.Count} records...");

            foreach (var record in dynamoEvent.Records)
            {
                Console.WriteLine($"Event ID: {record.EventID}");
                Console.WriteLine($"Event Name: {record.EventName}");

                string streamRecordJson = SerializeObject(record.Dynamodb);
                Console.WriteLine($"DynamoDB Record:");
                Console.WriteLine(streamRecordJson);
            }

            Console.WriteLine("Stream processing complete.");
        }

        private string SerializeObject(object streamRecord)
        {
            using (var ms = new MemoryStream())
            {
                _jsonSerializer.Serialize(streamRecord, ms);
                return Encoding.UTF8.GetString(ms.ToArray());
            }
        }
    }
}
```

```
}
```

Replace the `Program.cs` in a .NET Core project with the above sample. For instructions, see [.NET Core CLI \(p. 303\)](#).

Python 3

The following example processes messages from DynamoDB, and logs their contents.

Example `ProcessDynamoDBStream.py`

```
from __future__ import print_function

def lambda_handler(event, context):
    for record in event['Records']:
        print(record['eventID'])
        print(record['eventName'])
    print('Successfully processed %s records.' % str(len(event['Records'])))
```

Zip up the sample code to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Python \(p. 251\)](#).

Go

The following example processes messages from DynamoDB, and logs their contents.

Example

```
import (
    "strings"

    "github.com/aws/aws-lambda-go/events"
)

func handleRequest(ctx context.Context, e events.DynamoDBEvent) {

    for _, record := range e.Records {
        fmt.Printf("Processing request data for event ID %s, type %s.\n", record.EventID,
        record.EventName)

        // Print new values for attributes of type String
        for name, value := range record.Change.NewImage {
            if value.DataType() == events.DataTypeString {
                fmt.Printf("Attribute name: %s, value: %s\n", name, value.String())
            }
        }
    }
}
```

Zip up the sample code to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Python \(p. 251\)](#).

AWS SAM Template for a DynamoDB Application

You can build this application using [AWS SAM](#). To learn more about creating AWS SAM templates, see [AWS SAM Template Basics](#) in the [AWS Serverless Application Model Developer Guide](#).

Below is a sample AWS SAM template for the [tutorial application \(p. 172\)](#). Copy the text below to a `.yaml` file and save it next to the ZIP package you created previously. Note that the `Handler` and

Runtime parameter values should match the ones you used when you created the function in the previous section.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  ProcessDynamoDBStream:
    Type: AWS::Serverless::Function
    Properties:
      Handler: handler
      Runtime: runtime
      Policies: AWSLambdaDynamoDBExecutionRole
      Events:
        Stream:
          Type: DynamoDB
          Properties:
            Stream: !GetAtt DynamoDBTable.StreamArn
            BatchSize: 100
            StartingPosition: TRIM_HORIZON

  DynamoDBTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        -AttributeName: id
        AttributeType: S
      KeySchema:
        -AttributeName: id
        KeyType: HASH
      ProvisionedThroughput:
        ReadCapacityUnits: 5
        WriteCapacityUnits: 5
      StreamSpecification:
        StreamViewType: NEW_IMAGE
```

For information on how to package and deploy your serverless application using the package and deploy commands, see [Deploying Serverless Applications](#) in the *AWS Serverless Application Model Developer Guide*.

Using AWS Lambda with Amazon Kinesis

You can use an AWS Lambda function to process records in an [Amazon Kinesis data stream](#). With Kinesis, you can collect data from many sources and process them with multiple consumers. Lambda supports standard data stream iterators and HTTP/2 stream consumers.

Lambda reads records from the data stream and invokes your function [synchronously \(p. 82\)](#) with an event that contains stream records. Lambda reads records in batches and invokes your function to process records from the batch.

Example Kinesis Record Event

```
{ "Records": [
  {
    "kinesis": {
      "kinesisSchemaVersion": "1.0",
      "partitionKey": "1",
      "sequenceNumber":
"49590338271490256608559692538361571095921575989136588898",
```

```
        "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
        "approximateArrivalTimestamp": 1545084650.987
    },
    "eventSource": "aws:kinesis",
    "eventVersion": "1.0",
    "eventID":
"shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
    "eventName": "aws:kinesis:record",
    "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
    "awsRegion": "us-east-2",
    "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"
},
{
    "kinesis": {
        "kinesisSchemaVersion": "1.0",
        "partitionKey": "1",
        "sequenceNumber":
"49590338271490256608559692540925702759324208523137515618",
        "data": "VGhpccyBpcyBvbmx5IGEgdGVzdC4=",
        "approximateArrivalTimestamp": 1545084711.166
    },
    "eventSource": "aws:kinesis",
    "eventVersion": "1.0",
    "eventID":
"shardId-000000000006:49590338271490256608559692540925702759324208523137515618",
    "eventName": "aws:kinesis:record",
    "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
    "awsRegion": "us-east-2",
    "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"
}
]
```

If you have multiple applications that are reading records from the same stream, you can use Kinesis stream consumers instead of standard iterators. Consumers have dedicated read throughput so they don't have to compete with other consumers of the same data. With consumers, Kinesis pushes records to Lambda over an HTTP/2 connection, which can also reduce latency between adding a record and function invocation.

If your function returns an error, Lambda retries the batch until processing succeeds or the data expires. Until the issue is resolved, no data in the shard is processed. To avoid stalled shards and potential data loss, make sure to handle and record processing errors in your code.

Topics

- [Configuring Your Data Stream and Function \(p. 181\)](#)
- [Creating an Event Source Mapping \(p. 182\)](#)
- [Event Source Mapping API \(p. 183\)](#)
- [Execution Role Permissions \(p. 184\)](#)
- [Amazon CloudWatch Metrics \(p. 184\)](#)
- [Tutorial: Using AWS Lambda with Amazon Kinesis \(p. 184\)](#)
- [Sample Function Code \(p. 188\)](#)
- [AWS SAM Template for a Kinesis Application \(p. 191\)](#)

Configuring Your Data Stream and Function

Your Lambda function is a consumer application for your data stream. It processes one batch of records at a time from each shard. You can map a Lambda function to a data stream (standard iterator), or to a consumer of a stream ([enhanced fan-out](#)).

For standard iterators, Lambda polls each shard in your Kinesis stream for records at a base rate of once per second. When more records are available, Lambda keeps processing batches until it receives a batch that's smaller than the configured maximum batch size. The function shares read throughput with other consumers of the shard.

To minimize latency and maximize read throughput, create a data stream consumer. Stream consumers get a dedicated connection to each shard that doesn't impact other applications reading from the stream. The dedicated throughput can help if you have many applications reading the same data, or if you're reprocessing a stream with large records.

Stream consumers use HTTP/2 to reduce latency by pushing records to Lambda over a long-lived connection and compressing request headers. You can create a stream consumer with the Kinesis [RegisterStreamConsumer API](#).

```
$ aws kinesis register-stream-consumer --consumer-name con1 \
--stream-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream
{
    "Consumer": {
        "ConsumerName": "con1",
        "ConsumerARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream/
consumer/con1:1540591608",
        "ConsumerStatus": "CREATING",
        "ConsumerCreationTimestamp": 1540591608.0
    }
}
```

To increase the speed at which your function processes records, add shards to your data stream. Lambda processes records in each shard in order, and stops processing additional records in a shard if your function returns an error. With more shards, there are more batches being processed at once, which lowers the impact of errors on concurrency.

Note

If the total size of the records in a batch exceeds the [payload limit \(p. 7\)](#), Lambda splits it into smaller batches for processing.

If your function can't scale up to handle one concurrent execution per shard, [request a limit increase \(p. 7\)](#) or [reserve concurrency \(p. 37\)](#) for your function. The concurrency available to your function should match or exceed the number of shards in your Kinesis data stream.

Creating an Event Source Mapping

Create an event source mapping to tell Lambda to send records from your data stream to a Lambda function. You can create multiple event source mappings to process the same data with multiple Lambda functions, or process items from multiple data streams with a single function.

To configure your function to read from Kinesis in the Lambda console, create a [Kinesis trigger](#).

To create a trigger

1. Open the Lambda console [Functions page](#).
2. Choose a function.
3. Under **Designer**, choose a trigger type to add a trigger to your function.
4. Under **Configure triggers**, configure the event source options and then choose **Add**.
5. Choose **Save**.

Lambda supports the following options for Kinesis event sources.

Event Source Options

- **Kinesis stream** – The Kinesis stream to read records from.
- **Consumer** (optional) – Use a stream consumer to read from the stream over a dedicated connection.
- **Batch size** – The number of records to read from a shard in each batch, up to 10,000. Lambda passes all of the records in the batch to the function in a single call, as long as the total size of the events doesn't exceed the payload limit of 6 MB.
- **Starting position** – Process only new records, all existing records, or records created after a certain date.
 - **Latest** – Process new records that are added to the stream.
 - **Trim horizon** – Process all records in the stream.
 - **At timestamp** – Process records starting from a specific time.

After processing any existing records, the function is caught up and continues to process new records.

- **Enabled** – Disable the event source to stop processing records. Lambda keeps track of the last record processed and resumes processing from that point when it's re-enabled.

To manage the event source configuration later, choose the trigger in the designer.

Event Source Mapping API

To create the event source mapping with the AWS CLI, use the [CreateEventSourceMapping \(p. 351\)](#) API. The following example uses the AWS CLI to map a function named `my-function` to a Kinesis data stream. The data stream is specified by an Amazon Resource Name (ARN), with a batch size of 500 hundred, starting from the timestamp in Unix time.

```
$ aws lambda create-event-source-mapping --function-name my-function --no-enabled \
--batch-size 500 --starting-position AT_TIMESTAMP --starting-position-timestamp 1541139109
\
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b1128",
    "BatchSize": 500,
    "EventSourceArn": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "LastModified": 1541139209.351,
    "LastProcessingResult": "No records processed",
    "State": "Creating",
    "StateTransitionReason": "User action"
}
```

To use a consumer, specify the consumer's ARN instead of the stream's ARN. The `--no-enabled` option creates the event source mapping without enabling it. To enable it, use `update-event-source-mapping`.

```
$ aws lambda update-event-source-mapping --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b1128 --enabled
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b1128",
    "BatchSize": 500,
    "EventSourceArn": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "LastModified": 1541190239.996,
    "LastProcessingResult": "No records processed",
    "State": "Enabling",
    "StateTransitionReason": "User action"
```

}

To delete the event source mapping, use `delete-event-source-mapping`.

```
$ aws lambda delete-event-source-mapping --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "EventSourceArn": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "LastModified": 1541190240.0,
    "LastProcessingResult": "No records processed",
    "State": "Deleting",
    "StateTransitionReason": "User action"
}
```

Execution Role Permissions

Lambda needs the following permissions to manage resources that are related to your Kinesis data stream. Add them to your function's [execution role \(p. 9\)](#).

- [kinesis:DescribeStream](#)
- [kinesis:DescribeStreamSummary](#)
- [kinesis:GetRecords](#)
- [kinesis:GetShardIterator](#)
- [kinesis>ListShards](#)
- [kinesis>ListStreams](#)
- [kinesis:SubscribeToShard](#)

The `AWSLambdaKinesisExecutionRole` managed policy includes these permissions. For more information, see [AWS Lambda Execution Role \(p. 9\)](#).

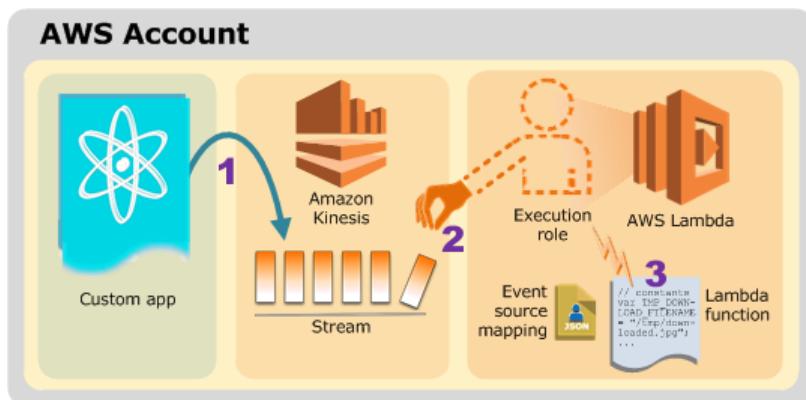
Amazon CloudWatch Metrics

Lambda emits the `IteratorAge` metric when your function finishes processing a batch of records. The metric indicates how old the last record in the batch was when processing finished. If your function is processing new events, you can use the iterator age to estimate the latency between when a record is added, and when the function processes it.

An increasing trend in iterator age can indicate issues with your function. For more information, see [Using Amazon CloudWatch \(p. 226\)](#).

Tutorial: Using AWS Lambda with Amazon Kinesis

In this tutorial, you create a Lambda function to consume events from a Kinesis stream. The following diagram illustrates the application flow:



1. Custom app writes records to the stream.
2. AWS Lambda polls the stream and, when it detects new records in the stream, invokes your Lambda function.
3. AWS Lambda executes the Lambda function by assuming the execution role you specified at the time you created the Lambda function.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Getting Started with AWS Lambda \(p. 3\)](#) to create your first Lambda function.

To follow the procedures in this guide, you will need a command line terminal or shell to run commands. Commands are shown in listings preceded by a prompt symbol (\$) and the name of the current directory, when appropriate:

```
~/lambda-project$ this is a command  
this is output
```

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

On Linux and macOS, use your preferred shell and package manager. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the Execution Role

Create the [execution role \(p. 9\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – AWS Lambda.**
 - **Permissions – AWSLambdaKinesisExecutionRole.**
 - **Role name – lambda-kinesis-role.**

The **AWSLambdaKinesisExecutionRole** policy has the permissions that the function needs to read items from Kinesis and write logs to CloudWatch Logs.

Create the Function

The following example code receives a Kinesis event input and processes the messages that it contains. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Note

For sample code in other languages, see [Sample Function Code \(p. 188\)](#).

Example index.js

```
console.log('Loading function');

exports.handler = function(event, context) {
    //console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        // Kinesis data is base64 encoded so decode here
        var payload = new Buffer(record.kinesis.data, 'base64').toString('ascii');
        console.log('Decoded payload:', payload);
    });
};
```

To create the function

1. Copy the sample code into a file named `index.js`.
2. Create a deployment package.

```
$ zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command.

```
$ aws lambda create-function --function-name ProcessKinesisRecords \
--zip-file file:///function.zip --handler index.handler --runtime nodejs8.10 \
--role arn:aws:iam::123456789012:role/lambda-kinesis-role
```

Test the Lambda Function

Invoke your Lambda function manually using the `invoke` AWS Lambda CLI command and a sample Kinesis event.

To test the Lambda function

1. Copy the following JSON into a file and save it as `input.txt`.

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
"49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
"shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "approximateArrivalTimestamp": 1545084650.987
        }
    ]
}
```

```
        "eventName": "aws:kinesis:record",
        "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-kinesis-role",
        "awsRegion": "us-east-2",
        "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
    }
}
```

2. Use the `invoke` command to send the event to the function.

```
$ aws lambda invoke --function-name ProcessKinesisRecords --payload file://input.txt
out.txt
```

The response is saved to `out.txt`.

Create a Kinesis Stream

Use the `create-stream` command to create a stream.

```
$ aws kinesis create-stream --stream-name lambda-stream --shard-count 1
```

Run the following `describe-stream` command to get the stream ARN.

```
$ aws kinesis describe-stream --stream-name lambda-stream
{
    "StreamDescription": {
        "Shards": [
            {
                "ShardId": "shardId-000000000000",
                "HashKeyRange": {
                    "StartingHashKey": "0",
                    "EndingHashKey": "340282366920746074317682119384634633455"
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber":
"49591073947768692513481539594623130411957558361251844610"
                }
            }
        ],
        "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/lambda-stream",
        "StreamName": "lambda-stream",
        "StreamStatus": "ACTIVE",
        "RetentionPeriodHours": 24,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
        "KeyId": null,
        "StreamCreationTimestamp": 1544828156.0
    }
}
```

You use the stream ARN in the next step to associate the stream with your Lambda function.

Add an Event Source in AWS Lambda

Run the following AWS CLI `add-event-source` command.

```
$ aws lambda create-event-source-mapping --function-name ProcessKinesisRecords \
--event-source arn:aws:kinesis:us-west-2:123456789012:stream/lambda-stream \
--batch-size 100 --starting-position LATEST
```

Note the mapping ID for later use. You can get a list of event source mappings by running the `list-event-source-mappings` command.

```
$ aws lambda list-event-source-mappings --function-name ProcessKinesisRecords \
--event-source arn:aws:kinesis:us-west-2:123456789012:stream/lambda-stream
```

In the response, you can verify the status value is enabled. Event source mappings can be disabled to pause polling temporarily losing any records.

Test the Setup

To test the event source mapping, add event records to your Kinesis stream. The `--data` value is a string that the CLI encodes to base64 prior to sending it to Kinesis. You can run the same command more than once to add multiple records to the stream.

```
$ aws kinesis put-record --stream-name lambda-stream --partition-key 1 \
--data "Hello, this is a test."
```

Lambda uses the execution role to read records from the stream. Then it invokes your Lambda function, passing in batches of records. The function decodes data from each record and logs it, sending the output to CloudWatch Logs. View the logs in the [CloudWatch console](#).

Sample Function Code

Sample code is available for the following languages.

Topics

- [Node.js 8 \(p. 188\)](#)
- [Java 8 \(p. 189\)](#)
- [C# \(p. 189\)](#)
- [Python 3 \(p. 190\)](#)
- [Go \(p. 190\)](#)

Node.js 8

The following example code receives a Kinesis event input and processes the messages that it contains. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Example index.js

```
console.log('Loading function');

exports.handler = function(event, context) {
    //console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        // Kinesis data is base64 encoded so decode here
        var payload = new Buffer(record.kinesis.data, 'base64').toString('ascii');
        console.log('Decoded payload:', payload);
    });
};
```

Zip up the sample code to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Node.js \(p. 241\)](#).

Java 8

The following is example Java code that receives Kinesis event record data as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

In the code, `recordHandler` is the handler. The handler uses the predefined `KinesisEvent` class that is defined in the `aws-lambda-java-events` library.

Example ProcessKinesisEvents.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.KinesisEventRecord;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent, Void>{
    @Override
    public Void handleRequest(KinesisEvent event, Context context)
    {
        for(KinesisEventRecord rec : event.getRecords()) {
            System.out.println(new String(rec.getKinesis().getData().array()));
        }
        return null;
    }
}
```

If the handler returns normally without exceptions, Lambda considers the input batch of records as processed successfully and begins reading new records in the stream. If the handler throws an exception, Lambda considers the input batch of records as not processed and invokes the function with the same batch of records again.

Dependencies

- `aws-lambda-java-core`
- `aws-lambda-java-events`
- `aws-java-sdk-s3`

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Java \(p. 264\)](#).

C#

The following is example C# code that receives Kinesis event record data as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

In the code, `HandleKinesisRecord` is the handler. The handler uses the predefined `KinesisEvent` class that is defined in the `Amazon.Lambda.KinesisEvents` library.

Example ProcessingKinesisEvents.cs

```
using System;
using System.IO;
using System.Text;
```

```
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;

namespace KinesisStreams
{
    public class KinesisSample
    {
        [LambdaSerializer(typeof(JsonSerializer))]
        public void HandleKinesisRecord(KinesisEvent kinesisEvent)
        {
            Console.WriteLine($"Beginning to process {kinesisEvent.Records.Count} records...");

            foreach (var record in kinesisEvent.Records)
            {
                Console.WriteLine($"Event ID: {record.EventId}");
                Console.WriteLine($"Event Name: {record.EventName}");

                string recordData = GetRecordContents(record.Kinesis);
                Console.WriteLine($"Record Data:");
                Console.WriteLine(recordData);
            }
            Console.WriteLine("Stream processing complete.");
        }

        private string GetRecordContents(KinesisEvent.Record streamRecord)
        {
            using (var reader = new StreamReader(streamRecord.Data, Encoding.ASCII))
            {
                return reader.ReadToEnd();
            }
        }
    }
}
```

Replace the `Program.cs` in a .NET Core project with the above sample. For instructions, see [.NET Core CLI \(p. 303\)](#).

Python 3

The following is example Python code that receives Kinesis event record data as input and processes it. For illustration, the code writes to some of the incoming event data to CloudWatch Logs.

Example `ProcessKinesisRecords.py`

```
from __future__ import print_function
import json
import base64
def lambda_handler(event, context):
    for record in event['Records']:
        #Kinesis data is base64 encoded so decode here
        payload=base64.b64decode(record["kinesis"]["data"])
        print("Decoded payload: " + str(payload))
```

Zip up the sample code to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Python \(p. 251\)](#).

Go

The following is example Go code that receives Kinesis event record data as input and processes it. For illustration, the code writes to some of the incoming event data to CloudWatch Logs.

Example ProcessKinesisRecords.go

```
import (
    "strings"
    "github.com/aws/aws-lambda-go/events"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) {
    for _, record := range kinesisEvent.Records {
        kinesisRecord := record.Kinesis
        dataBytes := kinesisRecord.Data
        dataText := string(dataBytes)

        fmt.Printf("%s Data = %s \n", record.EventName, dataText)
    }
}
```

Build the executable with `go build` and create a deployment package. For instructions, see [AWS Lambda Deployment Package in Go \(p. 290\)](#).

AWS SAM Template for a Kinesis Application

You can build this application using [AWS SAM](#). To learn more about creating AWS SAM templates, see [AWS SAM Template Basics](#) in the [AWS Serverless Application Model Developer Guide](#).

Below is a sample AWS SAM template for the Lambda application from the [tutorial \(p. 184\)](#). The function and handler in the template are for the Node.js code. If you use a different code sample, update the values accordingly.

Example template.yaml - Kinesis Stream

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  LambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs8.10
      Timeout: 10
      Tracing: Active
    Events:
      Stream:
        Type: Kinesis
        Properties:
          Stream: !GetAtt KinesisStream.Arn
          BatchSize: 100
          StartingPosition: LATEST
  KinesisStream:
    Type: AWS::Kinesis::Stream
    Properties:
      ShardCount: 1
Outputs:
  FunctionName:
    Description: "Function name"
    Value: !Ref LambdaFunction
  StreamARN:
    Description: "Stream ARN"
    Value: !GetAtt KinesisStream.Arn
```

The template creates a Lambda function, a Kinesis stream, and an event source mapping. The event source mapping reads from the stream and invokes the function.

To use an [HTTP/2 stream consumer \(p. 181\)](#), create the consumer in the template and configure the event source mapping to read from the consumer instead of from the stream.

Example template.yaml - Kinesis Stream Consumer

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: A function that processes data from a Kinesis stream.
Resources:
  kinesisprocessrecordpython:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs8.10
      Timeout: 10
      Tracing: Active
      Events:
        Stream:
          Type: Kinesis
          Properties:
            Stream: !GetAtt StreamConsumer.ConsumerARN
            StartingPosition: LATEST
            BatchSize: 100
      KinesisStream:
        Type: "AWS::Kinesis::Stream"
        Properties:
          ShardCount: 1
  StreamConsumer:
    Type: "AWS::Kinesis::StreamConsumer"
    Properties:
      StreamARN: !GetAtt KinesisStream.Arn
      ConsumerName: "TestConsumer"
Outputs:
  FunctionName:
    Description: "Function name"
    Value: !Ref LambdaFunction
  StreamARN:
    Description: "Stream ARN"
    Value: !GetAtt KinesisStream.Arn
  ConsumerARN:
    Description: "Stream consumer ARN"
    Value: !GetAtt StreamConsumer.ConsumerARN
```

For information on how to package and deploy your serverless application using the package and deploy commands, see [Deploying Serverless Applications](#) in the *AWS Serverless Application Model Developer Guide*.

Using AWS Lambda with Amazon Kinesis Data Firehose

Amazon Kinesis Data Firehose captures, transforms, and loads streaming data into downstream services such as Kinesis Data Analytics or Amazon S3. You can write Lambda functions to request additional, customized processing of the data before it is sent downstream.

Example Amazon Kinesis Data Firehose Message Event

```
{  
  "invocationId": "invoked123",
```

```

"deliveryStreamArn": "aws:lambda:events",
"region": "us-west-2",
"records": [
    {
        "data": "SGVsbG8gV29ybGQ=",
        "recordId": "record1",
        "approximateArrivalTimestamp": 1510772160000,
        "kinesisRecordMetadata": {
            "shardId": "shardId-000000000000",
            "partitionKey": "4d1ad2b9-24f8-4b9d-a088-76e9947c317a",
            "approximateArrivalTimestamp": "2012-04-23T18:25:43.511Z",
            "sequenceNumber": "49546986683135544286507457936321625675700192471156785154",
            "subsequenceNumber": ""
        }
    },
    {
        "data": "SGVsbG8gV29ybGQ=",
        "recordId": "record2",
        "approximateArrivalTimestamp": 151077216000,
        "kinesisRecordMetadata": {
            "shardId": "shardId-000000000001",
            "partitionKey": "4d1ad2b9-24f8-4b9d-a088-76e9947c318a",
            "approximateArrivalTimestamp": "2012-04-23T19:25:43.511Z",
            "sequenceNumber": "49546986683135544286507457936321625675700192471156785155",
            "subsequenceNumber": ""
        }
    }
]
}

```

For more information, see [Amazon Kinesis Data Firehose Data Transformation](#) in the Kinesis Data Firehose Developer Guide.

Using AWS Lambda with Amazon Lex

Amazon Lex is an AWS service for building conversational interfaces into applications using voice and text. Amazon Lex provides pre-build integration with AWS Lambda, allowing you to create Lambda functions for use as code hook with your Amazon Lex bot. In your intent configuration, you can identify your Lambda function to perform initialization/validation, fulfillment, or both.

Example Amazon Lex Message Event

```
{
    "messageVersion": "1.0",
    "invocationSource": "FulfillmentCodeHook",
    "userId": "ABCD1234",
    "sessionAttributes": {
        "key1": "value1",
        "key2": "value2",
    },
    "bot": {
        "name": "my-bot",
        "alias": "prod",
        "version": "1"
    },
    "outputDialogMode": "Text",
    "currentIntent": {
        "name": "my-intent",
        "slots": {
            "slot-name": "value",
            "slot-name": "value",
        }
    }
}
```

```
        "slot-name": "value"
    },
    "confirmationStatus": "Confirmed"
}
```

For more information, see [Using Lambda Functions](#). For an example use case, see [Exercise 1: Create Amazon Lex Bot Using a Blueprint](#).

Using AWS Lambda with Amazon S3

Amazon S3 can publish events (for example, when an object is created in a bucket) to AWS Lambda and invoke your Lambda function by passing the event data as a parameter. This integration enables you to write Lambda functions that process Amazon S3 events. In Amazon S3, you add bucket notification configuration that identifies the type of event that you want Amazon S3 to publish and the Lambda function that you want to invoke.

Important

If your Lambda function uses the same bucket that triggers it, it could cause the function to execute in a loop. For example, if the bucket triggers a function each time an object is uploaded, and the function uploads an object to the bucket, then the function indirectly triggers itself. To avoid this, use two buckets, or configure the trigger to only apply to a prefix used for incoming objects.

Example Amazon S3 Message Event

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-west-2",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "AIDAJDPLRKLG7UEXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "C3D13FE58DE4C810",
        "x-amz-id-2": "FMyUVURIY8/IgAtTv8xRjskZQpcIZ9KG4V5Wp6S7S/JRWeUWerMUE5JgHvANOjpD"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "sourcebucket",
          "ownerIdentity": {
            "principalId": "A3NL1KOZZKExample"
          },
          "arn": "arn:aws:s3:::sourcebucket"
        },
        "object": {
          "key": "HappyFace.jpg",
          "size": 1024,
          "eTag": "d41d8cd98f00b204e9800998ecf8427e",
          "versionId": "096fKKXTRTt13on89fVO.nfljtsv6qko"
        }
      }
    }
}
```

```
}
```

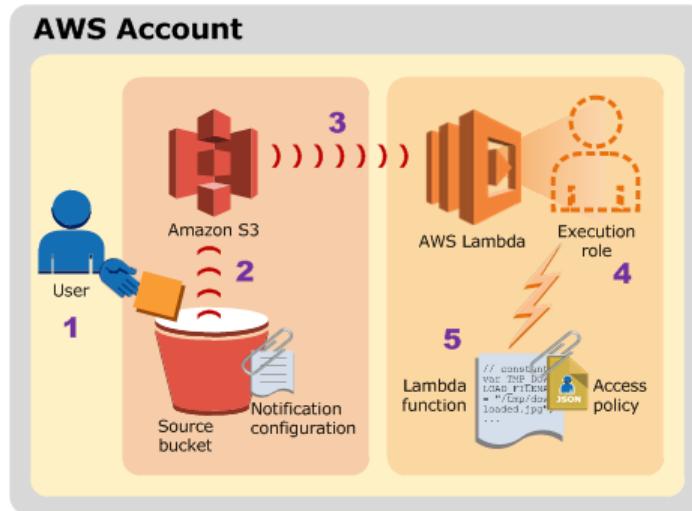
Note the following about how the Amazon S3 and AWS Lambda integration works:

- **Non-stream based (async) model** – This is a model (see [AWS Lambda Event Source Mapping \(p. 82\)](#)), where Amazon S3 monitors a bucket and invokes the Lambda function by passing the event data as a parameter. In a push model, you maintain event source mapping within Amazon S3 using the bucket notification configuration. In the configuration, you specify the event types that you want Amazon S3 to monitor and which AWS Lambda function you want Amazon S3 to invoke. For more information, see [Configuring Amazon S3 Event Notifications](#) in the *Amazon Simple Storage Service Developer Guide*.
- **Asynchronous invocation** – AWS Lambda invokes a Lambda function using the Event invocation type (asynchronous invocation). For more information about invocation types, see [Invocation Types \(p. 82\)](#).
- **Event structure** – The event your Lambda function receives is for a single object and it provides information, such as the bucket name and object key name.

Note that there are two types of permissions policies that you work with when you set up the end-to-end experience:

- **Permissions for your Lambda function** – Regardless of what invokes a Lambda function, AWS Lambda executes the function by assuming the IAM role (execution role) that you specify at the time you create the Lambda function. Using the permissions policy associated with this role, you grant your Lambda function the permissions that it needs. For example, if your Lambda function needs to read an object, you grant permissions for the relevant Amazon S3 actions in the permissions policy. For more information, see [AWS Lambda Execution Role \(p. 9\)](#).
- **Permissions for Amazon S3 to invoke your Lambda function** – Amazon S3 cannot invoke your Lambda function without your permission. You grant this permission via the permissions policy associated with the Lambda function.

The following diagram summarizes the flow:



1. User uploads an object to an S3 bucket (object-created event).
2. Amazon S3 detects the object-created event.
3. Amazon S3 invokes a Lambda function that is specified in the bucket notification configuration.
4. AWS Lambda executes the Lambda function by assuming the execution role that you specified at the time you created the Lambda function.

5. The Lambda function executes.

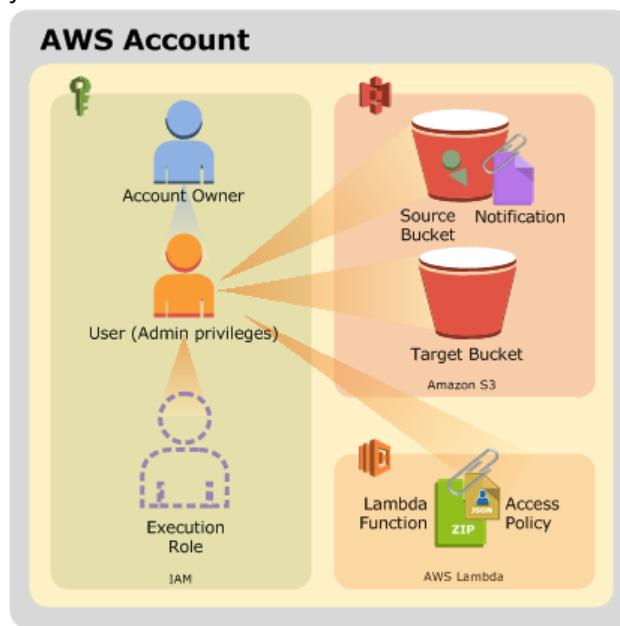
Topics

- [Tutorial: Using AWS Lambda with Amazon S3 \(p. 196\)](#)
- [Sample Amazon Simple Storage Service Function Code \(p. 202\)](#)
- [AWS SAM Template for an Amazon S3 Application \(p. 208\)](#)

Tutorial: Using AWS Lambda with Amazon S3

Suppose you want to create a thumbnail for each image file that is uploaded to a bucket. You can create a Lambda function (`CreateThumbnail`) that Amazon S3 can invoke when objects are created. Then, the Lambda function can read the image object from the source bucket and create a thumbnail image target bucket.

Upon completing this tutorial, you will have the following Amazon S3, Lambda, and IAM resources in your account:



Lambda Resources

- A Lambda function.
- An access policy associated with your Lambda function that grants Amazon S3 permission to invoke the Lambda function.

IAM Resources

- An execution role that grants permissions that your Lambda function needs through the permissions policy associated with this role.

Amazon S3 Resources

- A source bucket with a notification configuration that invokes the Lambda function.
- A target bucket where the function saves resized images.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Getting Started with AWS Lambda \(p. 3\)](#) to create your first Lambda function.

To follow the procedures in this guide, you will need a command line terminal or shell to run commands. Commands are shown in listings preceded by a prompt symbol (\$) and the name of the current directory, when appropriate:

```
~/lambda-project$ this is a command  
this is output
```

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

On Linux and macOS, use your preferred shell and package manager. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Install NPM to manage the function's dependencies.

Create the Execution Role

Create the [execution role \(p. 9\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – AWS Lambda.**
 - **Permissions – AWSLambdaExecute.**
 - **Role name – lambda-s3-role.**

The **AWSLambdaExecute** policy has the permissions that the function needs to manage objects in Amazon S3 and write logs to CloudWatch Logs.

Create Buckets and Upload a Sample Object

Follow the steps to create buckets and upload an object.

1. Open the Amazon S3 console.
2. Create two buckets. The target bucket name must be **source** followed by **resized**, where **source** is the name of the bucket you want to use for the source. For example, `mybucket` and `mybucketresized`.
3. In the source bucket, upload a .jpg object, `HappyFace.jpg`.

When you invoke the Lambda function manually before you connect to Amazon S3, you pass sample event data to the function that specifies the source bucket and `HappyFace.jpg` as the newly created object so you need to create this sample object first.

Create the Function

The following example code receives an Amazon S3 event input and processes the message that it contains. It resizes an image in the source bucket and saves the output to the target bucket.

Note

For sample code in other languages, see [Sample Amazon Simple Storage Service Function Code \(p. 202\)](#).

Example index.js

```
// dependencies
var async = require('async');
var AWS = require('aws-sdk');
var gm = require('gm')
    .subClass({ imageMagick: true }); // Enable ImageMagick integration.
var util = require('util');

// constants
var MAX_WIDTH  = 100;
var MAX_HEIGHT = 100;

// get reference to S3 client
var s3 = new AWS.S3();

exports.handler = function(event, context, callback) {
    // Read options from the event.
    console.log("Reading options from event:\n", util.inspect(event, {depth: 5}));
    var srcBucket = event.Records[0].s3.bucket.name;
    // Object key may have spaces or unicode non-ASCII characters.
    var srcKey    =
        decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, " "));
    var dstBucket = srcBucket + "resized";
    var dstKey    = "resized-" + srcKey;

    // Sanity check: validate that source and destination are different buckets.
    if (srcBucket == dstBucket) {
        callback("Source and destination buckets are the same.");
        return;
    }

    // Infer the image type.
    var typeMatch = srcKey.match(/\.([^.]*$)/);
    if (!typeMatch) {
        callback("Could not determine the image type.");
        return;
    }
    var imageType = typeMatch[1];
    if (imageType != "jpg" && imageType != "png") {
        callback('Unsupported image type: ${imageType}');
        return;
    }

    // Download the image from S3, transform, and upload to a different S3 bucket.
    async.waterfall([
        function download(next) {
            // Download the image from S3 into a buffer.
            s3.getObject({
                Bucket: srcBucket,
                Key: srcKey
            },
            next);
        },
        function transform(response, next) {
            gm(response.Body).size(function(err, size) {
                // Infer the scaling factor to avoid stretching the image unnaturally.
                var scalingFactor = Math.min(
                    MAX_WIDTH / size.width,
                    MAX_HEIGHT / size.height
                );
            });
        }
    ],
    function done(err) {
        if (err) {
            callback("Error processing image: " + err.message);
        } else {
            callback(null, "Image processed successfully!");
        }
    }
});
```

```

        var width  = scalingFactor * size.width;
        var height = scalingFactor * size.height;

        // Transform the image buffer in memory.
        this.resize(width, height)
            .toBuffer(imageType, function(err, buffer) {
                if (err) {
                    next(err);
                } else {
                    next(null, response.ContentType, buffer);
                }
            });
    },
    function upload(contentType, data, next) {
        // Stream the transformed image to a different S3 bucket.
        s3.putObject({
            Bucket: dstBucket,
            Key: dstKey,
            Body: data,
            ContentType: contentType
        },
        next);
    }
], function (err) {
    if (err) {
        console.error(
            'Unable to resize ' + srcBucket + '/' + srcKey +
            ' and upload to ' + dstBucket + '/' + dstKey +
            ' due to an error: ' + err
        );
    } else {
        console.log(
            'Successfully resized ' + srcBucket + '/' + srcKey +
            ' and uploaded to ' + dstBucket + '/' + dstKey
        );
    }
}

callback(null, "message");
);
);
};

```

Review the preceding code and note the following:

- The function knows the source bucket name and the key name of the object from the event data it receives as parameters. If the object is a .jpg, the code creates a thumbnail and saves it to the target bucket.
- The code assumes that the destination bucket exists and its name is a concatenation of the source bucket name followed by the string `resized`. For example, if the source bucket identified in the event data is `examplebucket`, the code assumes you have an `examplebucketresized` destination bucket.
- For the thumbnail it creates, the code derives its key name as the concatenation of the string `resized-` followed by the source object key name. For example, if the source object key is `sample.jpg`, the code creates a thumbnail object that has the key `resized-sample.jpg`.

The deployment package is a .zip file containing your Lambda function code and dependencies.

To create a deployment package

1. Save the function code as `index.js` in a folder named `lambda-s3`.
2. Install the GraphicsMagick and Async libraries with NPM.

```
lambda-s3$ npm install async gm
```

After you complete this step, you will have the following folder structure:

```
lambda-s3
|- index.js
|- /node_modules/gm
# /node_modules/async
```

3. Create a deployment package with the function code and dependencies.

```
lambda-s3$ zip -r function.zip .
```

To create the function

- Create a Lambda function with the `create-function` command.

```
$ aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs8.10 \
--timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/lambda-s3-role
```

The preceding command specifies a 10-second timeout value as the function configuration. Depending on the size of objects you upload, you might need to increase the timeout value using the following AWS CLI command.

```
$ aws lambda update-function-configuration --function-name CreateThumbnail --timeout 30
```

Test the Lambda Function

In this step, you invoke the Lambda function manually using sample Amazon S3 event data.

To test the Lambda function

1. Save the following Amazon S3 sample event data in a file and save it as `inputFile.txt`. You need to update the JSON by providing your `sourcebucket` name and a .jpg object key.

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-west-2",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "AIDAJDPLRKLG7UEXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "C3D13FE58DE4C810",
        "x-amz-id-2": "FMyUVURIY8/IgAtTv8xRjskZQpcIZ9KG4V5Wp6S7S/JRWeUWerMUE5JgHvANOjpD"
      }
    }
  ]
}
```

```
"s3":{  
    "s3SchemaVersion":"1.0",  
    "configurationId":"testConfigRule",  
    "bucket":{  
        "name":"sourcebucket",  
        "ownerIdentity":{  
            "principalId":"A3NL1KOZZKExample"  
        },  
        "arn":"arn:aws:s3:::sourcebucket"  
    },  
    "object":{  
        "key":"HappyFace.jpg",  
        "size":1024,  
        "eTag":"d41d8cd98f00b204e9800998ecf8427e",  
        "versionId":"096fKKXTRTl3on89fVO.nfljtsv6qko"  
    }  
}  
}  
]
```

2. Run the following Lambda CLI invoke command to invoke the function. Note that the command requests asynchronous execution. You can optionally invoke it synchronously by specifying RequestResponse as the invocation-type parameter value.

```
$ aws lambda invoke --function-name CreateThumbnail --invocation-type Event \  
--payload file://inputfile.txt outputfile.txt
```

3. Verify that the thumbnail was created in the target bucket.

Configure Amazon S3 to Publish Events

In this step, you add the remaining configuration so that Amazon S3 can publish object-created events to AWS Lambda and invoke your Lambda function. You do the following in this step:

- Add permissions to the Lambda function access policy to allow Amazon S3 to invoke the function.
- Add notification configuration to your source bucket. In the notification configuration, you provide the following:
 - Event type for which you want Amazon S3 to publish events. For this tutorial, you specify the s3:ObjectCreated:* event type so that Amazon S3 publishes events when objects are created.
 - Lambda function to invoke.

To add permissions to the function policy

1. Run the following Lambda CLI add-permission command to grant Amazon S3 service principal (s3.amazonaws.com) permissions to perform the lambda:InvokeFunction action. Note that permission is granted to Amazon S3 to invoke the function only if the following conditions are met:
 - An object-created event is detected on a specific bucket.
 - The bucket is owned by a specific AWS account. If a bucket owner deletes a bucket, some other AWS account can create a bucket with the same name. This condition ensures that only a specific AWS account can invoke your Lambda function.

```
$ aws lambda add-permission --function-name CreateThumbnail --principal  
s3.amazonaws.com \  
--statement-id some-unique-id --action "lambda:InvokeFunction" \  
--source-arn arn:aws:s3:::sourcebucket \  
--region us-west-2
```

```
--source-account bucket-owner-account-id
```

2. Verify the function's access policy by running the AWS CLI get-policy command.

```
$ aws lambda get-policy --function-name function-name
```

Add notification configuration on the source bucket to request Amazon S3 to publish object-created events to Lambda.

To configure notifications

1. Open the [Amazon S3 console](#).
2. Choose the source bucket.
3. Choose **Properties**.
4. Under **Events**, configure a notification with the following settings.
 - **Name** – `lambda-trigger`.
 - **Events** – `ObjectCreate (All)`.
 - **Send to** – `Lambda function`.
 - **Lambda** – `CreateThumbnail`.

For more information on event configuration, see [Enabling Event Notifications](#) in the *Amazon Simple Storage Service Console User Guide*.

Test the Setup

Now you can test the setup as follows:

1. Upload .jpg or .png objects to the source bucket using the Amazon S3 console.
2. Verify that the thumbnail was created in the target bucket using the `CreateThumbnail` function.
3. View logs in the CloudWatch console.

Sample Amazon Simple Storage Service Function Code

Sample code is available for the following languages.

Topics

- [Node.js 8 \(p. 202\)](#)
- [Java 8 \(p. 205\)](#)
- [Python 3 \(p. 207\)](#)

Node.js 8

The following example code receives an Amazon S3 event input and processes the message that it contains. It resizes an image in the source bucket and saves the output to the target bucket.

Example index.js

```
var async = require('async');
```

```
var AWS = require('aws-sdk');
var gm = require('gm')
    .subClass({ imageMagick: true }); // Enable ImageMagick integration.
var util = require('util');

var MAX_WIDTH  = 100;
var MAX_HEIGHT = 100;

var s3 = new AWS.S3();

exports.handler = function(event, context, callback) {
    // Read options from the event.
    console.log("Reading options from event:\n", util.inspect(event, {depth: 5}));
    var srcBucket = event.Records[0].s3.bucket.name;
    // Object key may have spaces or unicode non-ASCII characters.
    var srcKey    =
        decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, " "));
    var dstBucket = srcBucket + "resized";
    var dstKey    = "resized-" + srcKey;

    // Sanity check: validate that source and destination are different buckets.
    if (srcBucket == dstBucket) {
        callback("Source and destination buckets are the same.");
        return;
    }

    // Infer the image type.
    var typeMatch = srcKey.match(/\.([^.]*$)/);
    if (!typeMatch) {
        callback("Could not determine the image type.");
        return;
    }
    var imageType = typeMatch[1];
    if (imageType != "jpg" && imageType != "png") {
        callback('Unsupported image type: ${imageType}');
        return;
    }

    // Download the image from S3, transform, and upload to a different S3 bucket.
    async.waterfall([
        function download(next) {
            // Download the image from S3 into a buffer.
            s3.getObject({
                Bucket: srcBucket,
                Key: srcKey
            },
            next);
        },
        function transform(response, next) {
            gm(response.Body).size(function(err, size) {
                // Infer the scaling factor to avoid stretching the image unnaturally.
                var scalingFactor = Math.min(
                    MAX_WIDTH / size.width,
                    MAX_HEIGHT / size.height
                );
                var width  = scalingFactor * size.width;
                var height = scalingFactor * size.height;

                // Transform the image buffer in memory.
                this.resize(width, height)
                    .toBuffer(imageType, function(err, buffer) {
                        if (err) {
                            next(err);
                        } else {
                            next(null, response.ContentType, buffer);
                        }
                    })
            });
        }
    ],
    function upload(next) {
        s3.upload({
            Bucket: dstBucket,
            Key: dstKey,
            Body: buffer
        }, next);
    }
], callback);
}
```

```
        });
    },
    function upload(contentType, data, next) {
        // Stream the transformed image to a different S3 bucket.
        s3.putObject({
            Bucket: dstBucket,
            Key: dstKey,
            Body: data,
            ContentType: contentType
        },
        next);
    }
], function (err) {
    if (err) {
        console.error(
            'Unable to resize ' + srcBucket + '/' + srcKey +
            ' and upload to ' + dstBucket + '/' + dstKey +
            ' due to an error: ' + err
        );
    } else {
        console.log(
            'Successfully resized ' + srcBucket + '/' + srcKey +
            ' and uploaded to ' + dstBucket + '/' + dstKey
        );
    }
    callback(null, "message");
});
});
```

The deployment package is a .zip file containing your Lambda function code and dependencies.

To create a deployment package

1. Create a folder (`examplefolder`), and then create a subfolder (`node_modules`).
2. Install the Node.js platform. For more information, see the [Node.js](#) website.
3. Install dependencies. The code examples use the following libraries:
 - AWS SDK for JavaScript in Node.js
 - gm, GraphicsMagick for node.js
 - Async utility module

The AWS Lambda runtime already has the AWS SDK for JavaScript in Node.js, so you only need to install the other libraries. Open a command prompt, navigate to the `examplefolder`, and install the libraries using the `npm` command, which is part of Node.js.

```
$ npm install async gm
```

4. Save the sample code to a file named `index.js`.
5. Review the preceding code and note the following:
 - The function knows the source bucket name and the key name of the object from the event data it receives as parameters. If the object is a .jpg, the code creates a thumbnail and saves it to the target bucket.
 - The code assumes that the destination bucket exists and its name is a concatenation of the source bucket name followed by the string `resized`. For example, if the source bucket identified in

the event data is `examplebucket`, the code assumes you have an `examplebucketresized` destination bucket.

- For the thumbnail it creates, the code derives its key name as the concatenation of the string `resized-` followed by the source object key name. For example, if the source object key is `sample.jpg`, the code creates a thumbnail object that has the key `resized-sample.jpg`.
6. Save the file as `index.js` in `examplefolder`. After you complete this step, you will have the following folder structure:

```
index.js
/node_modules/gm
/node_modules/async
```

7. Zip the `index.js` file and the `node_modules` folder as `CreateThumbnail.zip`.

Java 8

The following is example Java code that reads incoming Amazon S3 events and creates a thumbnail. Note that it implements the `RequestHandler` interface provided in the `aws-lambda-java-core` library. Therefore, at the time you create a Lambda function you specify the class as the handler (that is, `example.S3EventProcessorCreateThumbnail`). For more information about using interfaces to provide a handler, see [Leveraging Predefined Interfaces for Creating Handler \(Java\) \(p. 277\)](#).

The `S3Event` type that the handler uses as the input type is one of the predefined classes in the `aws-lambda-java-events` library that provides methods for you to easily read information from the incoming Amazon S3 event. The handler returns a string as output.

Example `S3EventProcessorCreateThumbnail.java`

```
package example;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.image.BufferedImage;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URLDecoder;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.imageio.ImageIO;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3Client;
import com.amazonaws.services.s3.event.S3EventNotification.S3EventNotificationRecord;
import com.amazonaws.services.s3.model.GetObjectRequest;
import com.amazonaws.services.s3.model.ObjectMetadata;
import com.amazonaws.services.s3.model.S3Object;

public class S3EventProcessorCreateThumbnail implements
    RequestHandler<S3Event, String> {
    private static final float MAX_WIDTH = 100;
    private static final float MAX_HEIGHT = 100;
    private final String JPG_TYPE = (String) "jpg";
```

```
private final String JPG_MIME = (String) "image/jpeg";
private final String PNG_TYPE = (String) "png";
private final String PNG_MIME = (String) "image/png";

public String handleRequest(S3Event s3event, Context context) {
    try {
        S3EventNotificationRecord record = s3event.getRecords().get(0);

        String srcBucket = record.getS3().getBucket().getName();
        // Object key may have spaces or unicode non-ASCII characters.
        String srcKey = record.getS3().getObject().getKey()
            .replace('+', ' ');
        srcKey = URLDecoder.decode(srcKey, "UTF-8");

        String dstBucket = srcBucket + "resized";
        String dstKey = "resized-" + srcKey;

        // Sanity check: validate that source and destination are different
        // buckets.
        if (srcBucket.equals(dstBucket)) {
            System.out
                .println("Destination bucket must not match source bucket.");
            return "";
        }

        // Infer the image type.
        Matcher matcher = Pattern.compile(".*\\.(\\[^\\.]*)").matcher(srcKey);
        if (!matcher.matches()) {
            System.out.println("Unable to infer image type for key "
                + srcKey);
            return "";
        }
        String imageType = matcher.group(1);
        if (!(JPG_TYPE.equals(imageType)) && !(PNG_TYPE.equals(imageType))) {
            System.out.println("Skipping non-image " + srcKey);
            return "";
        }

        // Download the image from S3 into a stream
        AmazonS3 s3Client = new AmazonS3Client();
        S3Object s3Object = s3Client.getObject(new GetObjectRequest(
            srcBucket, srcKey));
        InputStream objectData = s3Object.getObjectContent();

        // Read the source image
        BufferedImage srcImage = ImageIO.read(objectData);
        int srcHeight = srcImage.getHeight();
        int srcWidth = srcImage.getWidth();
        // Infer the scaling factor to avoid stretching the image
        // unnaturally
        float scalingFactor = Math.min(MAX_WIDTH / srcWidth, MAX_HEIGHT
            / srcHeight);
        int width = (int) (scalingFactor * srcWidth);
        int height = (int) (scalingFactor * srcHeight);

        BufferedImage resizedImage = new BufferedImage(width, height,
            BufferedImage.TYPE_INT_RGB);
        Graphics2D g = resizedImage.createGraphics();
        // Fill with white before applying semi-transparent (alpha) images
        g.setPaint(Color.white);
        g.fillRect(0, 0, width, height);
        // Simple bilinear resize
        g.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
            RenderingHints.VALUE_INTERPOLATION_BILINEAR);
        g.drawImage(srcImage, 0, 0, width, height, null);
        g.dispose();
```

```
// Re-encode image to target format
ByteArrayOutputStream os = new ByteArrayOutputStream();
ImageIO.write(resizedImage, imageType, os);
InputStream is = new ByteArrayInputStream(os.toByteArray());
// Set Content-Length and Content-Type
ObjectMetadata meta = new ObjectMetadata();
meta.setContentLength(os.size());
if (JPG_TYPE.equals(imageType)) {
    meta.setContentType(JPG_MIME);
}
if (PNG_TYPE.equals(imageType)) {
    meta.setContentType(PNG_MIME);
}

// Uploading to S3 destination bucket
System.out.println("Writing to: " + dstBucket + "/" + dstKey);
s3Client.putObject(dstBucket, dstKey, is, meta);
System.out.println("Successfully resized " + srcBucket + "/"
    + srcKey + " and uploaded to " + dstBucket + "/" + dstKey);
return "Ok";
} catch (IOException e) {
    throw new RuntimeException(e);
}
}
}
```

Amazon S3 invokes your Lambda function using the `Event` invocation type, where AWS Lambda executes the code asynchronously. What you return does not matter. However, in this case we are implementing an interface that requires us to specify a return type, so in this example the handler uses `String` as the return type.

Dependencies

- `aws-lambda-java-core`
- `aws-lambda-java-events`
- `aws-java-sdk-s3`

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Java \(p. 264\)](#).

Python 3

The following example code receives an Amazon S3 event input and processes the message that it contains. It resizes an image in the source bucket and saves the output to the target bucket.

Example CreateThumbnail.py

```
import boto3
import os
import sys
import uuid
from PIL import Image
import PIL.Image

s3_client = boto3.client('s3')

def resize_image(image_path, resized_path):
    with Image.open(image_path) as image:
```

```
image.thumbnail(tuple(x / 2 for x in image.size))
image.save(resized_path)

def handler(event, context):
    for record in event['Records']:
        bucket = record['s3']['bucket']['name']
        key = record['s3']['object']['key']
        download_path = '/tmp/{}/{}/'.format(uuid.uuid4(), key)
        upload_path = '/tmp/resized-{}'.format(key)

        s3_client.download_file(bucket, key, download_path)
        resize_image(download_path, upload_path)
        s3_client.upload_file(upload_path, '{}resized'.format(bucket), key)
```

To create a deployment package

1. Copy the sample code into a file named `CreateThumbnail.py`.
2. Create a virtual environment.

```
$ virtualenv ~/shrink_venv
```

```
$ source ~/shrink_venv/bin/activate
```

3. Install libraries in the virtual environment

```
$ pip install Pillow
```

```
$ pip install boto3
```

4. Add the contents of lib and lib64 site-packages to your .zip file.

```
$ cd $VIRTUAL_ENV/lib/python3.7/site-packages
```

```
$ zip -r ~/CreateThumbnail.zip .
```

5. Add your python code to the .zip file

```
$ cd ~
```

```
$ zip -g CreateThumbnail.zip CreateThumbnail.py
```

AWS SAM Template for an Amazon S3 Application

You can build this application using [AWS SAM](#). To learn more about creating AWS SAM templates, see [AWS SAM Template Basics](#) in the [AWS Serverless Application Model Developer Guide](#).

Below is a sample AWS SAM template for the Lambda application from the [tutorial \(p. 196\)](#). Copy the text below to a .yaml file and save it next to the ZIP package you created previously. Note that the Handler and Runtime parameter values should match the ones you used when you created the function in the previous section.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  CreateThumbnail:
    Type: AWS::Serverless::Function
    Properties:
      Handler: handler
```

```
Runtime: runtime
Timeout: 60
Policies: AWSLambdaExecute
Events:
  CreateThumbnailEvent:
    Type: S3
    Properties:
      Bucket: !Ref SrcBucket
      Events: s3:ObjectCreated:*
SrcBucket:
  Type: AWS::S3::Bucket
```

For information on how to package and deploy your serverless application using the package and deploy commands, see [Deploying Serverless Applications](#) in the *AWS Serverless Application Model Developer Guide*.

Using AWS Lambda with Amazon SES

When you use Amazon SES to receive messages, you can configure Amazon SES to call your Lambda function when messages arrive. The service can then invoke your Lambda function by passing in the incoming email event, which in reality is an Amazon SES message in an Amazon SNS event, as a parameter.

Example Amazon SES Message Event

```
{
  "Records": [
    {
      "eventVersion": "1.0",
      "ses": {
        "mail": {
          "commonHeaders": {
            "from": [
              "Jane Doe <janedoe@example.com>"
            ],
            "to": [
              "johndoe@example.com"
            ],
            "returnPath": "janedoe@example.com",
            "messageId": "<0123456789@example.com>",
            "date": "Wed, 7 Oct 2015 12:34:56 -0700",
            "subject": "Test Subject"
          },
          "source": "janedoe@example.com",
          "timestamp": "1970-01-01T00:00:00.000Z",
          "destination": [
            "johndoe@example.com"
          ],
          "headers": [
            {
              "name": "Return-Path",
              "value": "<janedoe@example.com>"
            },
            {
              "name": "Received",
              "value": "from mailer.example.com (mailer.example.com [203.0.113.1]) by inbound-smtp.us-west-2.amazonaws.com with SMTP id o3vrnil0e2ic for johndoe@example.com; Wed, 07 Oct 2015 12:34:56 +0000 (UTC)"
            },
            ...
          ]
        }
      }
    }
  ]
}
```

```
{
    "name": "DKIM-Signature",
    "value": "v=1; a=rsa-sha256; c=relaxed/relaxed; d=example.com;
s=example; h=mime-version:from:date:message-id:subject:to:content-type;
bh=jX3F0bCAI7sIbkHyy3mLYO28ieDQz2R0P8HwQkklFj4=; b=sQwJ+LMe9RjkesGu+vqU56asvMhrLRRYrWCbV"
},
{
    "name": "MIME-Version",
    "value": "1.0"
},
{
    "name": "From",
    "value": "Jane Doe <janedoe@example.com>"
},
{
    "name": "Date",
    "value": "Wed, 7 Oct 2015 12:34:56 -0700"
},
{
    "name": "Message-ID",
    "value": "<0123456789example.com>"
},
{
    "name": "Subject",
    "value": "Test Subject"
},
{
    "name": "To",
    "value": "johndoe@example.com"
},
{
    "name": "Content-Type",
    "value": "text/plain; charset=UTF-8"
}
],
"headersTruncated": false,
"messageId": "o3vrnil0e2ic28tr"
},
"receipt": {
    "recipients": [
        "johndoe@example.com"
    ],
    "timestamp": "1970-01-01T00:00:00.000Z",
    "spamVerdict": {
        "status": "PASS"
    },
    "dkimVerdict": {
        "status": "PASS"
    },
    "processingTimeMillis": 574,
    "action": {
        "type": "Lambda",
        "invocationType": "Event",
        "functionArn": "arn:aws:lambda:us-west-2:012345678912:function:Example"
    },
    "spfVerdict": {
        "status": "PASS"
    },
    "virusVerdict": {
        "status": "PASS"
    }
},
"eventSource": "aws:ses"
}
]
```

}

For more information, see [Lambda Action in the Amazon SES Developer Guide](#).

Using AWS Lambda with Amazon SNS

You can write Lambda functions to process Amazon Simple Notification Service notifications. When a message is published to an Amazon SNS topic, the service can invoke your Lambda function by passing the message payload as a parameter. Your Lambda function code can then process the event, for example publish the message to other Amazon SNS topics, or send the message to other AWS services.

When a user calls the SNS Publish API on a topic that your Lambda function is subscribed to, Amazon SNS will call Lambda to invoke your function asynchronously. Lambda will then return a delivery status. If there was an error calling Lambda, Amazon SNS will retry invoking the Lambda function up to three times. After three tries, if Amazon SNS still could not successfully invoke the Lambda function, then Amazon SNS will send a delivery status failure message to CloudWatch.

In order to perform cross account Amazon SNS deliveries to Lambda, you need to authorize your Lambda function to be invoked from Amazon SNS. In turn, Amazon SNS needs to allow the Lambda account to subscribe to the Amazon SNS topic. For example, if the Amazon SNS topic is in account A and the Lambda function is in account B, both accounts must grant permissions to the other to access their respective resources. Since not all the options for setting up cross-account permissions are available from the AWS console, you use the AWS CLI to set up the entire process.

Example Amazon SNS Message Event

```
{  
  "Records": [  
    {  
      "EventVersion": "1.0",  
      "EventSubscriptionArn": "arn:aws:sns:us-east-2:123456789012:test-lambda:21be56ed-a058-49f5-8c98-aedd2564c486",  
      "EventSource": "aws:sns",  
      "Sns": {  
        "SignatureVersion": "1",  
        "Timestamp": "1970-01-01T00:00:00.000Z",  
        "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkaI6RibDsvpi+tE/1+82j...65r==",  
        "SigningCertUrl": "https://sns.us-east-2.amazonaws.com/SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",  
        "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",  
        "Message": "Hello from SNS!",  
        "MessageAttributes": {  
          "Test": {  
            "Type": "String",  
            "Value": "TestString"  
          },  
          "TestBinary": {  
            "Type": "Binary",  
            "Value": "TestBinary"  
          }  
        },  
        "Type": "Notification",  
        "UnsubscribeUrl": "https://sns.us-east-2.amazonaws.com/?Action=Unsubscribe&SubscriptionArn=arn:aws:sns:us-east-2:123456789012:test-lambda:21be56ed-a058-49f5-8c98-aedd2564c486",  
        "TopicArn": "topicarn",  
        "Subject": "TestInvoke"  
      }  
    }  
  ]
```

}

You configure the event source mapping in Amazon SNS via topic subscription configuration. For more information, see [Invoking Lambda functions using Amazon SNS notifications](#) in the *Amazon Simple Notification Service Developer Guide*.

Topics

- [Tutorial: Using AWS Lambda with Amazon Simple Notification Service \(p. 212\)](#)
- [Sample Function Code \(p. 214\)](#)

Tutorial: Using AWS Lambda with Amazon Simple Notification Service

You can use a Lambda function in one AWS account to subscribe to an Amazon SNS topic in a separate AWS account. In this tutorial, you use the AWS Command Line Interface to perform AWS Lambda operations such as creating a Lambda function, creating an Amazon SNS topic and granting permissions to allow these two resources to access each other.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Getting Started with AWS Lambda \(p. 3\)](#) to create your first Lambda function.

To follow the procedures in this guide, you will need a command line terminal or shell to run commands. Commands are shown in listings preceded by a prompt symbol (\$) and the name of the current directory, when appropriate:

```
~/lambda-project$ this is a command  
this is output
```

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

On Linux and macOS, use your preferred shell and package manager. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

In the tutorial, you use two accounts. The AWS CLI commands illustrate this by using two [named profiles](#), each configured for use with a different account. If you use profiles with different names, or the default profile and one named profile, modify the commands as needed.

Create an Amazon SNS Topic

From account A, create the source Amazon SNS topic.

```
$ aws sns create-topic --name lambda-x-account --profile accountA
```

Note the topic ARN that is returned by the command. You will need it when you add permissions to the Lambda function to subscribe to the topic.

Create the Execution Role

From account B, create the [execution role \(p. 9\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – AWS Lambda.**
 - **Permissions – AWSLambdaBasicExecutionRole.**
 - **Role name – lambda-sns-role.**

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

Create a Lambda Function

From account B, create the function that processes events from Amazon SNS. The following example code receives an Amazon SNS event input and processes the messages that it contains. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Note

For sample code in other languages, see [Sample Function Code \(p. 214\)](#).

Example index.js

```
console.log('Loading function');

exports.handler = function(event, context, callback) {
// console.log('Received event:', JSON.stringify(event, null, 4));

    var message = event.Records[0].Sns.Message;
    console.log('Message received from SNS:', message);
    callback(null, "Success");
};
```

To create the function

1. Copy the sample code into a file named `index.js`.
2. Create a deployment package.

```
$ zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command.

```
$ aws lambda create-function --function-name SNS-X-Account \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs8.10 \
--role arn:aws:iam::01234567891B:role/service-role/lambda-sns-execution-role \
--timeout 60 --profile accountB
```

Note the function ARN that is returned by the command. You will need it when you add permissions to allow Amazon SNS to invoke your function.

Set Up Cross-Account Permissions

From account A, grant permission to account B to subscribe to the topic:

```
$ aws sns add-permission --label lambda-access --aws-account-id 12345678901B \
--topic-arn arn:aws:sns:us-east-2:12345678901A:lambda-x-account \
--action-name Subscribe ListSubscriptionsByTopic Receive --profile accountA
```

From account B, add the Lambda permission to allow invocation from Amazon SNS.

```
$ aws lambda add-permission --function-name SNS-X-Account \
--source-arn arn:aws:sns:us-east-2:12345678901A:lambda-x-account \
--statement-id sns-x-account --action "lambda:InvokeFunction" \
--principal sns.amazonaws.com --profile accountB
{
    "Statement": "{\"Condition\": {\"ArnLike\": {\"AWS:SourceArn\": \"arn:aws:lambda:us-east-2:12345678901B:function:SNS-X-Account\"}}, \"Action\": [\"lambda:InvokeFunction\"], \"Resource\": \"arn:aws:lambda:us-east-2:01234567891A:function:SNS-X-Account\", \"Effect\": \"Allow\", \"Principal\": {\"Service\": \"sns.amazonaws.com\"}, \"Sid\": \"sns-x-account1\"}"
}
```

Do not use the --source-account parameter to add a source account to the Lambda policy when adding the policy. Source account is not supported for Amazon SNS event sources and will result in access being denied.

Create a Subscription

From account B, subscribe the Lambda function to the topic. When a message is sent to the lambda-x-account topic in account A, Amazon SNS invokes the SNS-X-Account function in account B.

```
$ aws sns subscribe --protocol lambda \
--topic-arn arn:aws:sns:us-east-2:12345678901A:lambda-x-account \
--notification-endpoint arn:aws:lambda:us-east-2:12345678901B:function:SNS-X-Account \
--profile accountB
{
    "SubscriptionArn": "arn:aws:sns:us-east-2:12345678901A:lambda-x-account:5d906xxxx-7c8x-45dx-a9dx-0484e31c98xx"
}
```

The output contains the ARN of the topic subscription.

Test Subscription

From account A, test the subscription. Type Hello World into a text file and save it as `message.txt`. Then run the following command:

```
$ aws sns publish --message file://message.txt --subject Test \
--topic-arn arn:aws:sns:us-east-2:12345678901A:lambda-x-account \
--profile accountA
```

This will return a message id with a unique identifier, indicating the message has been accepted by the Amazon SNS service. Amazon SNS will then attempt to deliver it to the topic's subscribers. Alternatively, you could supply a JSON string directly to the `message` parameter, but using a text file allows for line breaks in the message.

To learn more about Amazon SNS, see [What is Amazon Simple Notification Service](#).

Sample Function Code

Sample code is available for the following languages.

Topics

- [Node.js 8 \(p. 215\)](#)
- [Java 8 \(p. 215\)](#)
- [Go \(p. 216\)](#)
- [Python 3 \(p. 216\)](#)

Node.js 8

The following example processes messages from Amazon SNS, and logs their contents.

Example index.js

```
console.log('Loading function');

exports.handler = function(event, context, callback) {
// console.log('Received event:', JSON.stringify(event, null, 4));

    var message = event.Records[0].Sns.Message;
    console.log('Message received from SNS:', message);
    callback(null, "Success");
};
```

Zip up the sample code to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Node.js \(p. 241\)](#).

Java 8

The following example processes messages from Amazon SNS, and logs their contents.

Example LambdaWithSNS.java

```
package example;

import java.text.SimpleDateFormat;
import java.util.Calendar;

import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;

public class LogEvent implements RequestHandler<SNSEvent, Object> {
    public Object handleRequest(SNSEvent request, Context context){
        String timeStamp = new SimpleDateFormat("yyyy-MM-
dd_HH:mm:ss").format(Calendar.getInstance().getTime());
        context.getLogger().log("Invocation started: " + timeStamp);
        context.getLogger().log(request.getRecords().get(0).getSNS().getMessage());

        timeStamp = new SimpleDateFormat("yyyy-MM-
dd_HH:mm:ss").format(Calendar.getInstance().getTime());
        context.getLogger().log("Invocation completed: " + timeStamp);
        return null;
    }
}
```

Dependencies

- [aws-lambda-java-core](#)
- [aws-lambda-java-events](#)

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Java \(p. 264\)](#).

Go

The following example processes messages from Amazon SNS, and logs their contents.

Example lambda_handler.go

```
package main

import (
    "context"
    "fmt"
    "github.com/aws/aws-lambda-go/lambda"
    "github.com/aws/aws-lambda-go/events"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
    for _, record := range snsEvent.Records {
        snsRecord := record.SNS
        fmt.Printf("[%s %s] Message = %s \n", record.EventSource, snsRecord.Timestamp,
snsRecord.Message)
    }
}

func main() {
    lambda.Start(handler)
}
```

Build the executable with `go build` and create a deployment package. For instructions, see [AWS Lambda Deployment Package in Go \(p. 290\)](#).

Python 3

The following example processes messages from Amazon SNS, and logs their contents.

Example lambda_handler.py

```
from __future__ import print_function
import json
print('Loading function')

def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))
    message = event['Records'][0]['Sns']['Message']
    print("From SNS: " + message)
    return message
```

Zip up the sample code to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Python \(p. 251\)](#).

Using AWS Lambda with Amazon SQS

You can use an AWS Lambda function to process messages in a [standard Amazon Simple Queue Service \(Amazon SQS\) queue](#). With Amazon SQS, you can offload tasks from one component of your application by sending them to a queue and processing them asynchronously.

Lambda polls the queue and invokes your function [synchronously \(p. 82\)](#) with an event that contains queue messages. Lambda reads messages in batches and invokes your function once for each batch. When your function successfully processes a batch, Lambda deletes its messages from the queue.

Example Amazon SQS Message Event

```
{  
    "Records": [  
        {  
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",  
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",  
            "body": "test",  
            "attributes": {  
                "ApproximateReceiveCount": "1",  
                "SentTimestamp": "1545082649183",  
                "SenderId": "AIDAIEQZJOL023YYJ4VO",  
                "ApproximateFirstReceiveTimestamp": "1545082649185"  
            },  
            "messageAttributes": {},  
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",  
            "eventSource": "aws:sqs",  
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",  
            "awsRegion": "us-east-2"  
        },  
        {  
            "messageId": "2e1424d4-f796-459a-8184-9c92662be6da",  
            "receiptHandle": "AQEBzWwaftRI0KuVm4tP+/7q1rGgNqicHq...",  
            "body": "test",  
            "attributes": {  
                "ApproximateReceiveCount": "1",  
                "SentTimestamp": "1545082650636",  
                "SenderId": "AIDAIEQZJOL023YYJ4VO",  
                "ApproximateFirstReceiveTimestamp": "1545082650649"  
            },  
            "messageAttributes": {},  
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",  
            "eventSource": "aws:sqs",  
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",  
            "awsRegion": "us-east-2"  
        }  
    ]  
}
```

Lambda uses [long polling](#) to poll a queue until it becomes active. When messages are available, Lambda increases the rate at which it reads batches, and invokes your function until it reaches a concurrency limit. For more information on how Lambda scales to process messages in your Amazon SQS queue, see [Understanding Scaling Behavior \(p. 86\)](#).

When Lambda reads a message from the queue, it stays in the queue but becomes hidden until Lambda deletes it. If your function returns an error, or doesn't finish processing before the queue's [visibility timeout](#), it becomes visible again. Then Lambda sends it to your Lambda function again. All messages in a failed batch return to the queue, so your function code must be able to process the same message multiple times without side effects.

Sections

- [Configuring a Queue for Use With Lambda \(p. 218\)](#)
- [Configuring a Queue as an Event Source \(p. 218\)](#)
- [Execution Role Permissions \(p. 218\)](#)
- [Tutorial: Using AWS Lambda with Amazon Simple Queue Service \(p. 219\)](#)
- [Sample Amazon SQS Function Code \(p. 222\)](#)

- [AWS SAM Template for an Amazon SQS Application \(p. 224\)](#)

Configuring a Queue for Use With Lambda

Create a standard Amazon SQS queue to serve as an event source for your Lambda function. Then configure the queue to allow time for your Lambda function to process each batch of events—and for Lambda to retry in response to throttling errors as it scales up.

To allow your function time to process each batch of records, set the source queue's visibility timeout to at least 6 times the [timeout \(p. 35\)](#) that you configure on your function. The extra time allows for Lambda to retry if your function execution is throttled while your function is processing a previous batch.

If a message fails processing multiple times, Amazon SQS can send it to a [dead letter queue](#). Configure a dead letter queue on your source queue to retain messages that failed processing for troubleshooting. Set the `maxReceiveCount` on the queue's redrive policy to at least 5 to avoid sending messages to the dead letter queue due to throttling.

Configuring a Queue as an Event Source

Create an event source mapping to tell Lambda to send items from your queue to a Lambda function. You can create multiple event source mappings to process items from multiple queues with a single function. When Lambda invokes the target function, the event can contain multiple items, up to a configurable maximum batch size.

To add an event source mapping for an Amazon SQS queue

1. Open the Lambda console [Functions page](#).
2. Choose a function.
3. Under **Add triggers**, choose **SQS**.
4. Under **Configure triggers**, configure the event source.
 - **SQS queue** – Specify the source queue.
 - **Batch size** – Specify the maximum number of items to read from the queue and send to your function, in a single invocation.
 - **Enabled** – Clear the check box to disable the event source.
5. Choose **Add**.
6. Choose **Save**.

Configure your function timeout to allow enough time to process an entire batch of items. If items take a long time to process, choose a smaller batch size. A large batch size can improve efficiency for workloads that are very fast or have a lot of overhead. However, if your function returns an error, all items in the batch return to the queue. If you configure [reserved concurrency \(p. 37\)](#) on your function, set a minimum of 5 concurrent executions to reduce the chance of throttling errors when Lambda invokes your function.

To configure an event source with the Lambda API or the AWS SDK, use the [CreateEventSourceMapping \(p. 351\)](#) and [UpdateEventSourceMapping \(p. 452\)](#) actions.

Execution Role Permissions

Lambda needs the following permissions to manage messages in your Amazon SQS queue. Add them to your function's [execution role \(p. 9\)](#).

- `sqs:ReceiveMessage`
- `sqs:DeleteMessage`

- [`sq:GetQueueAttributes`](#)

For more information, see [AWS Lambda Execution Role \(p. 9\)](#).

Tutorial: Using AWS Lambda with Amazon Simple Queue Service

In this tutorial, you create a Lambda function to consume messages from an [Amazon SQS](#) queue.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Getting Started with AWS Lambda \(p. 3\)](#) to create your first Lambda function.

To follow the procedures in this guide, you will need a command line terminal or shell to run commands. Commands are shown in listings preceded by a prompt symbol (\$) and the name of the current directory, when appropriate:

```
~/lambda-project$ this is a command  
this is output
```

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

On Linux and macOS, use your preferred shell and package manager. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the Execution Role

Create the [execution role \(p. 9\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – AWS Lambda.**
 - **Permissions – `AWSLambdaSQSQueueExecutionRole`.**
 - **Role name – `lambda-sqs-role`.**

The **`AWSLambdaSQSQueueExecutionRole`** policy has the permissions that the function needs to read items from Amazon SQS and write logs to CloudWatch Logs.

Create the Function

The following example code receives an Amazon SQS event input and processes the messages that it contains. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Note

For sample code in other languages, see [Sample Amazon SQS Function Code \(p. 222\)](#).

Example `index.js`

```
exports.handler = async function(event, context) {
```

```
event.Records.forEach(record => {
  const { body } = record;
  console.log(body);
});
return {};
}
```

To create the function

1. Copy the sample code into a file named `index.js`.
2. Create a deployment package.

```
$ zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command.

```
$ aws lambda create-function --function-name ProcessSQSRecord \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs8.10 \
--role arn:aws:iam::123456789012:role/lambda-sqs-role
```

Test the Function

Invoke your Lambda function manually using the `invoke` AWS Lambda CLI command and a sample Amazon Simple Queue Service event.

If the handler returns normally without exceptions, Lambda considers the message processed successfully and begins reading new messages in the queue. Once a message is processed successfully, it is automatically deleted from the queue. If the handler throws an exception, Lambda considers the input of messages as not processed and invokes the function with the same batch of messages.

1. Copy the following JSON into a file and save it as `input.txt`.

```
{
  "Records": [
    {
      "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
      "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
      "body": "test",
      "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIEQZJLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
      },
      "messageAttributes": {},
      "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
      "eventSource": "aws:sqs",
      "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
      "awsRegion": "us-east-2"
    }
  ]
}
```

2. Execute the following `invoke` command.

```
$ aws lambda invoke --invocation-type RequestResponse --function-name ProcessSQSRecord \
\ --payload file://input.txt outputfile.txt
```

The `invoke` command specifies `RequestResponse` as the invocation type, which requests synchronous execution. For more information, see [Invocation Types \(p. 82\)](#).

3. Verify the output in the `outputfile.txt` file.

Create an Amazon SQS Queue

Create an Amazon SQS queue that the Lambda function can use as an event source.

To create a queue

1. Sign in to the AWS Management Console and open the Amazon SQS console at <https://console.aws.amazon.com/sqs/>.
2. In the Amazon SQS console, create a queue.
3. Write down or otherwise record the identifying queue ARN (Amazon Resource Name). You need this in the next step when you associate the queue with your Lambda function.

Create an event source mapping in AWS Lambda. This event source mapping associates the Amazon SQS queue with your Lambda function. After you create this event source mapping, AWS Lambda starts polling the queue.

Test the end-to-end experience. As you perform queue updates, Amazon Simple Queue Service writes messages to the queue. AWS Lambda polls the queue, detects new records and executes your Lambda function on your behalf by passing events, in this case Amazon SQS messages, to the function.

Configure the Event Source

To create a mapping between the specified Amazon SQS queue and the Lambda function, run the following AWS CLI `create-event-source-mapping` command. After the command executes, write down or otherwise record the UUID. You'll need this UUID to refer to the event source mapping in any other commands, for example, if you choose to delete the event source mapping.

```
$ aws lambda create-event-source-mapping --function-name ProcessSQSRecord --batch-size 10 \
\ \
--event-source SQS-queue-arn
```

You can get the list of event source mappings by running the following command.

```
$ aws lambda list-event-source-mappings --function-name ProcessSQSRecord \
--event-source SQS-queue-arn
```

The list returns all of the event source mappings you created, and for each mapping it shows the `LastProcessingResult`, among other things. This field is used to provide an informative message if there are any problems. Values such as `No records processed` (indicates that AWS Lambda has not started polling or that there are no records in the queue) and `OK` (indicates AWS Lambda successfully read records from the queue and invoked your Lambda function) indicate that there are no issues. If there are issues, you receive an error message.

Test the Setup

Now you can test the setup as follows:

1. In the Amazon SQS console, send messages to the queue. Amazon SQS writes records of these actions to the queue.

2. AWS Lambda polls the queue and when it detects updates, it invokes your Lambda function by passing in the event data it finds in the queue.
3. Your function executes and creates logs in Amazon CloudWatch. You can verify the logs reported in the Amazon CloudWatch console.

Sample Amazon SQS Function Code

Sample code is available for the following languages.

Topics

- [Node.js \(p. 222\)](#)
- [Java \(p. 222\)](#)
- [C# \(p. 223\)](#)
- [Go \(p. 224\)](#)
- [Python \(p. 224\)](#)

Node.js

The following is example code that receives an Amazon SQS event message as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Example index.js (Node.js 8)

```
exports.handler = async function(event, context) {  
    event.Records.forEach(record => {  
        const { body } = record;  
        console.log(body);  
    });  
    return {};  
}
```

Example index.js (Node.js 6)

```
event.Records.forEach(function(record) {  
    var body = record.body;  
    console.log(body);  
});  
callback(null, "message");  
};
```

Zip up the sample code to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Node.js \(p. 241\)](#).

Java

The following is example Java code that receives an Amazon SQS event message as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

In the code, `handleRequest` is the handler. The handler uses the predefined `SQSEvent` class that is defined in the `aws-lambda-java-events` library.

Example ProcessSQSRecord.java

```
import com.amazonaws.services.lambda.runtime.Context;
```

```
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class ProcessSQSEvents implements RequestHandler<SQSEvent, Void>{
    @Override
    public Void handleRequest(SQSEvent event, Context context)
    {
        for(SQSMessage msg : event.getRecords()){
            System.out.println(new String(msg.getSQS().getBody()));
        }
        return null;
    }
}
```

Dependencies

- `aws-lambda-java-core`
- `aws-lambda-java-events`

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Java \(p. 264\)](#).

C#

The following is example C# code that receives an Amazon SQS event message as input and processes it. For illustration, the code writes some of the incoming event data to the console.

In the code, `handleRequest` is the handler. The handler uses the predefined `SQSEvent` class that is defined in the `AWS.Lambda.SQSEvents` library.

Example ProcessingSQSRecords.cs

```
[assembly: LambdaSerializer(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]

namespace SQSLambdaFunction
{
    public class SQSLambdaFunction
    {
        public string HandleSQSEvent(SQSEvent sqsEvent, ILambdaContext context)
        {
            Console.WriteLine($"Beginning to process {sqsEvent.Records.Count} records...");

            foreach (var record in sqsEvent.Records)
            {
                Console.WriteLine($"Message ID: {record.MessageId}");
                Console.WriteLine($"Event Source: {record.EventSource}");

                Console.WriteLine($"Record Body:");
                Console.WriteLine(record.Body);
            }

            Console.WriteLine("Processing complete.");

            return $"Processed {sqsEvent.Records.Count} records.";
        }
    }
}
```

Replace the `Program.cs` in a .NET Core project with the above sample. For instructions, see [.NET Core CLI \(p. 303\)](#).

Go

The following is example Go code that receives an Amazon SQS event message as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

In the code, `handler` is the handler. The handler uses the predefined `SQSEvent` class that is defined in the `aws-lambda-go-events` library.

Example ProcessSQSRecords.go

```
package main

import (
    "context"
    "fmt"

    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, sqsEvent events.SQSEvent) error {
    for _, message := range sqsEvent.Records {
        fmt.Printf("The message %s for event source %s = %s \n",
            message.MessageId,
            message.EventSource, message.Body)
    }

    return nil
}

func main() {
    lambda.Start(handler)
}
```

Build the executable with `go build` and create a deployment package. For instructions, see [AWS Lambda Deployment Package in Go \(p. 290\)](#).

Python

The following is example Python code that accepts an Amazon SQS record as input and processes it. For illustration, the code writes to some of the incoming event data to CloudWatch Logs.

Follow the instructions to create a AWS Lambda function deployment package.

Example ProcessSQSRecords.py

```
from __future__ import print_function

def lambda_handler(event, context):
    for record in event['Records']:
        print ("test")
        payload=record["body"]
        print(str(payload))
```

Zip up the sample code to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Python \(p. 251\)](#).

AWS SAM Template for an Amazon SQS Application

You can build this application using [AWS SAM](#). To learn more about creating AWS SAM templates, see [AWS SAM Template Basics](#) in the [AWS Serverless Application Model Developer Guide](#).

Below is a sample AWS SAM template for the Lambda application from the [tutorial \(p. 219\)](#). Copy the text below to a .yaml file and save it next to the ZIP package you created previously. Note that the Handler and Runtime parameter values should match the ones you used when you created the function in the previous section.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Example of processing messages on an SQS queue with Lambda
Resources:
  MySQSQueueFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: handler
      Runtime: runtime
      Events:
        MySQSEvent:
          Type: SQS
          Properties:
            Queue: !GetAtt MySqsQueue.Arn
            BatchSize: 10
  MySqsQueue:
    Type: AWS::SQS::Queue
    Properties:
      QueueName: MySqsQueue
```

For information on how to package and deploy your serverless application using the package and deploy commands, see [Deploying Serverless Applications](#) in the *AWS Serverless Application Model Developer Guide*.

Monitoring and Troubleshooting Lambda Applications

AWS Lambda will automatically track the behavior of your Lambda function invocations and provide feedback that you can monitor. In addition, it provides metrics that allows you to analyze the full function invocation spectrum, including event source integration and whether downstream resources perform as expected. The following sections provide guidance on the tools you can use to analyze your Lambda function invocation behavior:

Topics

- [Using Amazon CloudWatch \(p. 226\)](#)
- [Using AWS X-Ray \(p. 232\)](#)
- [Logging AWS Lambda API Calls with AWS CloudTrail \(p. 238\)](#)

Using Amazon CloudWatch

AWS Lambda automatically monitors Lambda functions on your behalf, reporting metrics through Amazon CloudWatch. To help you monitor your code as it executes, Lambda automatically tracks the number of requests, the execution duration per request, and the number of requests resulting in an error and publishes the associated CloudWatch metrics. You can leverage these metrics to set CloudWatch custom alarms. For more information about CloudWatch, see the [Amazon CloudWatch User Guide](#).

You can view request rates and error rates for each of your Lambda functions by using the AWS Lambda console, the CloudWatch console, and other Amazon Web Services (AWS) resources. The following topics describe Lambda CloudWatch metrics and how to access them.

- [Accessing Amazon CloudWatch Metrics for AWS Lambda \(p. 228\)](#)
- [AWS Lambda Metrics \(p. 229\)](#)

You can insert logging statements into your code to help you validate that your code is working as expected. Lambda automatically integrates with Amazon CloudWatch Logs and pushes all logs from your code to a CloudWatch Logs group associated with a Lambda function (/aws/lambda/<*function name*>). To learn more about log groups and accessing them through the CloudWatch console, see the [Monitoring System, Application, and Custom Log Files](#) in the [Amazon CloudWatch User Guide](#). For information about how to access CloudWatch log entries, see [Accessing Amazon CloudWatch Logs for AWS Lambda \(p. 229\)](#).

Note

If your Lambda function code is executing, but you don't see any log data being generated after several minutes, this could mean your execution role for the Lambda function did not grant permissions to write log data to CloudWatch Logs. For information about how to make sure that you have set up the execution role correctly to grant these permissions, see [AWS Lambda Execution Role \(p. 9\)](#).

AWS Lambda Troubleshooting Scenarios

This section describes examples of how to monitor and troubleshoot your Lambda functions using the logging and monitoring capabilities of CloudWatch.

Troubleshooting Scenario 1: Lambda Function Not Working as Expected

In this scenario, you have just finished [Tutorial: Using AWS Lambda with Amazon S3 \(p. 196\)](#). However, the Lambda function you created to upload a thumbnail image to Amazon S3 when you create an S3 object is not working as expected. When you upload objects to Amazon S3, you see that the thumbnail images are not being uploaded. You can troubleshoot this issue in the following ways.

To determine why your Lambda function is not working as expected

1. Check your code and verify that it is working correctly. An increased error rate would indicate that it is not.

You can test your code locally as you would any other Node.js function, or you can test it within the Lambda console using the console's test invoke functionality, or you can use the AWS CLI `Invoke` command. Each time the code is executed in response to an event, it writes a log entry into the log group associated with a Lambda function, which is `/aws/lambda/<function name>`.

Following are some examples of errors that might show up in the logs:

- If you see a stack trace in your log, there is probably an error in your code. Review your code and debug the error that the stack trace refers to.
 - If you see a `permissions denied` error in the log, the IAM role you have provided as an execution role may not have the necessary permissions. Check the IAM role and verify that it has all of the necessary permissions to access any AWS resources that your code references. To ensure that you have correctly set up the execution role, see [AWS Lambda Execution Role \(p. 9\)](#).
 - If you see a `timeout exceeded` error in the log, your function was terminated because it did not return prior to the configured timeout. This may be because the timeout is too low, or the code is taking too long to execute.
 - If you see a `memory exceeded` error in the log, your memory setting is too low. Set it to a higher value. For information about memory size limits, see [CreateFunction \(p. 355\)](#).
2. Check your Lambda function and verify that it is receiving requests.

Even if your function code is working as expected and responding correctly to test invokes, the function may not be receiving requests from Amazon S3. If Amazon S3 is able to invoke the function, you should see an increase in your CloudWatch requests metrics. If you do not see an increase in your CloudWatch requests, check the access permissions policy associated with the function.

Troubleshooting Scenario 2: Increased Duration in Lambda Function Execution

In this scenario, you have just finished [Tutorial: Using AWS Lambda with Amazon S3 \(p. 196\)](#). However, the Lambda function you created to upload a thumbnail image to Amazon S3 when you create an S3 object is not working as expected. When you upload objects to Amazon S3, you can see that the thumbnail images are being uploaded, but your code is taking much longer to execute than expected. You can troubleshoot this issue in a couple of different ways. For example, you could monitor the duration metric for the Lambda function to see if the execution time is increasing. Or you could see an increase in the CloudWatch errors metric for the Lambda function, which might be due to timeout errors.

To determine why there is increased duration in the execution of a Lambda function

1. Test your code with different memory settings.

If your code is taking too long to execute, it could be that it does not have enough compute resources to execute its logic. Try increasing the memory allocated to your function and testing the code again, using the Lambda console's test invoke functionality. You can see the memory used, code execution time, and memory allocated in the function log entries. Changing the memory setting can change how you are charged for execution time. For information about pricing, see [AWS Lambda](#).

2. Use logs to investigate the source of the execution bottleneck

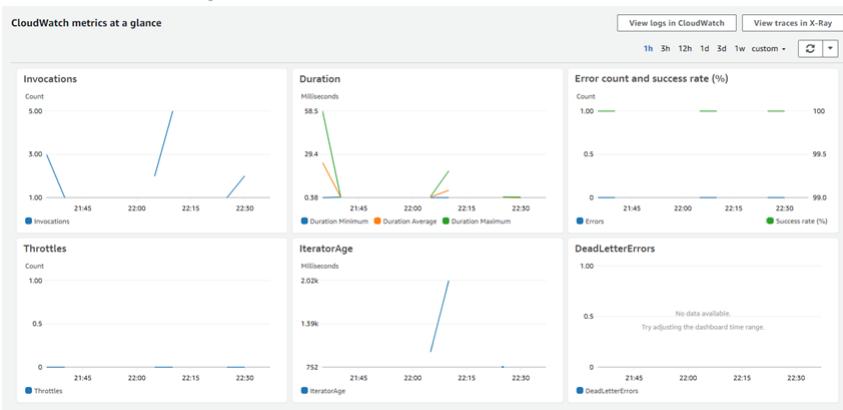
You can test your code locally, as you would with any other Node.js function, or you can test it within Lambda using the test invoke capability on the Lambda console, or using the `asyncInvoke` command by using AWS CLI. Each time the code is executed in response to an event, it writes a log entry into the log group associated with a Lambda function, which is named `aws/lambda/<function name>`. Add logging statements around various parts of your code, such as callouts to other services, to see how much time it takes to execute different parts of your code.

Accessing Amazon CloudWatch Metrics for AWS Lambda

AWS Lambda automatically monitors functions on your behalf, reporting metrics through Amazon CloudWatch. These metrics include total requests, duration, and error rates.

To access metrics using the Lambda console

1. Open the [Lambda console](#).
2. Open the Lambda console [Functions page](#).
3. Choose **Monitoring**.



The console provides the following graphs.

Lambda Monitoring Graphs

- **Invocations** – The number of times the function was invoked in each 5 minute period.
- **Duration** – Average, minimum, and maximum execution times.
- **Error count and success rate (%)** – The number of errors, and the percentage of executions that completed without error.
- **Throttles** – The number of times execution failed due to concurrency limits.
- **IteratorAge** – For stream event sources, the age of the last item in the batch when Lambda receives it and invokes the function.

- **DeadLetterErrors** – The number of events that Lambda attempted to write to a dead letter queue, but failed.

To see the definition of a graph in CloudWatch, choose **View in metrics** from the menu in the top right of the graph. For more information about the metrics that Lambda records, see [AWS Lambda Metrics \(p. 229\)](#).

Accessing Amazon CloudWatch Logs for AWS Lambda

AWS Lambda automatically monitors Lambda functions on your behalf, reporting metrics through Amazon CloudWatch. To help you troubleshoot failures in a function, Lambda logs all requests handled by your function and also automatically stores logs generated by your code through Amazon CloudWatch Logs.

You can insert logging statements into your code to help you validate that your code is working as expected. Lambda automatically integrates with CloudWatch Logs and pushes all logs from your code to a CloudWatch Logs group associated with a Lambda function, which is named `/aws/lambda/<function name>`. To learn more about log groups and accessing them through the CloudWatch console, see the [Monitoring System, Application, and Custom Log Files](#) in the [Amazon CloudWatch User Guide](#).

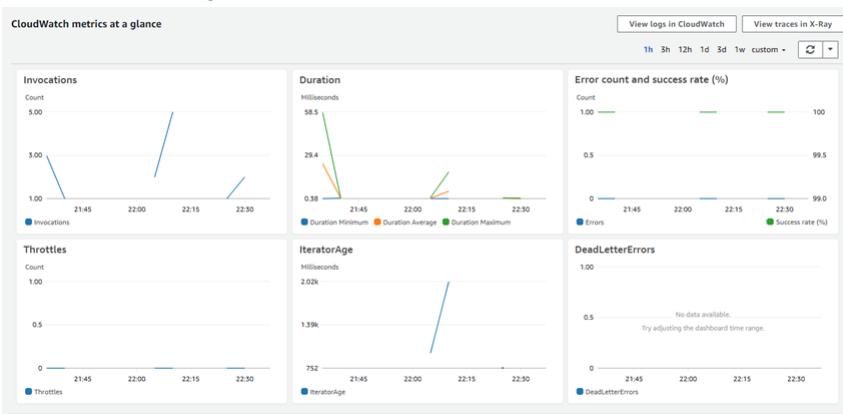
You can view logs for Lambda by using the Lambda console, the CloudWatch console, the AWS CLI, or the CloudWatch API. The following procedure show you how to view the logs by using the Lambda console.

Note

There is no additional charge for using Lambda logs; however, standard CloudWatch Logs charges apply. For more information, see [CloudWatch Pricing](#).

To view logs using the Lambda console

1. Open the [Lambda console](#).
2. Open the Lambda console [Functions page](#).
3. Choose **Monitoring**.



A graphical representation of the metrics for the Lambda function are shown.

4. Choose **View logs in CloudWatch**.

AWS Lambda Metrics

This topic describes the AWS Lambda namespace, metrics, and dimensions. AWS Lambda automatically monitors functions on your behalf, reporting metrics through Amazon CloudWatch. These metrics

include total invocations, errors, duration, throttles, DLQ errors and Iterator age for stream-based invocations.

CloudWatch is basically 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 time stamp, 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](#).

AWS Lambda CloudWatch Metrics

The AWS/Lambda namespace includes the following metrics.

Metric	Description
Invocations	<p>Measures the number of times a function is invoked in response to an event or invocation API call. This replaces the deprecated RequestCount metric. This includes successful and failed invocations, but does not include throttled attempts. This equals the billed requests for the function. Note that AWS Lambda only sends these metrics to CloudWatch if they have a nonzero value.</p> <p>Units: Count</p>
Errors	<p>Measures the number of invocations that failed due to errors in the function (response code 4XX). This replaces the deprecated ErrorCount metric. Failed invocations may trigger a retry attempt that succeeds. This includes:</p> <ul style="list-style-type: none"> • Handled exceptions (for example, context.fail(error)) • Unhandled exceptions causing the code to exit • Out of memory exceptions • Timeouts • Permissions errors <p>This does not include invocations that fail due to invocation rates exceeding default concurrent limits (error code 429) or failures due to internal service errors (error code 500).</p> <p>Units: Count</p>
DeadLetterErrors	<p>Incremented when Lambda is unable to write the failed event payload to your configured Dead Letter Queues. This could be due to the following:</p> <ul style="list-style-type: none"> • Permissions errors • Throttles from downstream services • Misconfigured resources • Timeouts <p>Units: Count</p>
Duration	Measures the elapsed wall clock time from when the function code starts executing as a result of an invocation to when it stops executing. The maximum data point value possible is the function timeout configuration.

Metric	Description
	The billed duration will be rounded up to the nearest 100 millisecond. Note that AWS Lambda only sends these metrics to CloudWatch if they have a nonzero value. Units: Milliseconds
Throttles	Measures the number of Lambda function invocation attempts that were throttled due to invocation rates exceeding the customer's concurrent limits (error code 429). Failed invocations may trigger a retry attempt that succeeds. Units: Count
IteratorAge	Emitted for stream-based invocations only (functions triggered by an Amazon DynamoDB stream or Kinesis stream). Measures the age of the last record for each batch of records processed. Age is the difference between the time Lambda received the batch, and the time the last record in the batch was written to the stream. Units: Milliseconds
ConcurrentExecutions	Emitted as an aggregate metric for all functions in the account, and for functions that have a custom concurrency limit specified. Not applicable for versions or aliases. Measures the sum of concurrent executions for a given function at a given point in time. Must be viewed as an average metric if aggregated across a time period. Units: Count
UnreservedConcurrency	Emitted as an aggregate metric for all functions in the account only. Not applicable for functions, versions, or aliases. Represents the sum of the concurrency of the functions that do not have a custom concurrency limit specified. Must be viewed as an average metric if aggregated across a time period. Units: Count

To access metrics using the CloudWatch console

1. Open the CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.
2. In the navigation pane, choose **Metrics**.
3. In the **CloudWatch Metrics by Category** pane, choose **Lambda Metrics**.

Errors/Invocations Ratio

When calculating the error rate on Lambda function invocations, it's important to distinguish between an invocation request and an actual invocation. It is possible for the error rate to exceed the number of billed Lambda function invocations. Lambda reports an invocation metric only if the Lambda function code is executed. If the invocation request yields a throttling or other initialization error that prevents the Lambda function code from being invoked, Lambda will report an error, but it does not log an invocation metric.

- Lambda emits `Invocations=1` when the function is executed. If the Lambda function is not executed, nothing is emitted.
- Lambda emits a data point for `Errors` for each invoke request. `Errors=0` means that there is no function execution error. `Errors=1` means that there is a function execution error.

- Lambda emits a data point for `Throttles` for each invoke request. `Throttles=0` means there is no invocation throttle. `Throttles=1` means there is an invocation throttle.

AWS Lambda CloudWatch Dimensions

You can use the dimensions in the following table to refine the metrics returned for your Lambda functions.

Dimension	Description
<code>FunctionName</code>	Filters the metric data by Lambda function.
<code>Resource</code>	Filters the metric data by Lambda function resource, such as function version or alias.
<code>ExecutedVersion</code>	Filters the metric data by Lambda function versions. This only applies to alias invocations.

Using AWS X-Ray

A typical Lambda-based application consists of one or more functions triggered by events such as object uploads to Amazon S3, Amazon SNS notifications, and API actions. Once triggered, those functions usually call downstream resources such as DynamoDB tables or Amazon S3 buckets, or make other API calls. AWS Lambda leverages Amazon CloudWatch to automatically emit metrics and logs for all invocations of your function. However, this mechanism might not be convenient for tracing the event source that invoked your Lambda function, or for tracing downstream calls that your function made. For a complete overview of how tracing works, see [AWS X-Ray](#).

Tracing Lambda-Based Applications with AWS X-Ray

AWS X-Ray is an AWS service that allows you to detect, analyze, and optimize performance issues with your AWS Lambda applications. X-Ray collects metadata from the Lambda service and any upstream or downstream services that make up your application. X-Ray uses this metadata to generate a detailed service graph that illustrates performance bottlenecks, latency spikes, and other issues that impact the performance of your Lambda application.

After using the [Lambda on the AWS X-Ray Service Map \(p. 232\)](#) to identify a problematic resource or component, you can zoom in and view a visual representation of the request. This visual representation covers the time from when an event source triggers a Lambda function until the function execution has completed. X-Ray provides you with a breakdown of your function's operations, such as information regarding downstream calls your Lambda function made to other services. In addition, X-Ray integration with Lambda provides you with visibility into the AWS Lambda service overhead. It does so by displaying specifics such as your request's dwell time and number of invocations.

Note

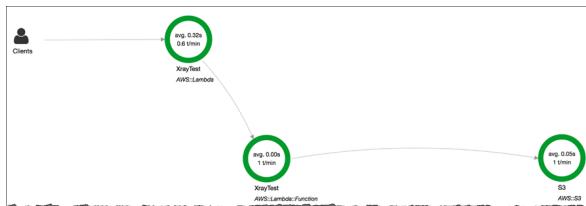
Only services that currently integrate with X-Ray show as standalone traces, outside of your Lambda trace. For a list of services that currently support X-Ray, see [Integrating AWS X-Ray with Other AWS Services](#).

Lambda on the AWS X-Ray Service Map

X-Ray displays three types of nodes on the service map for requests served by Lambda:

- **Lambda service (AWS::Lambda)** – This type of node represents the time the request spent in the Lambda service. Timing starts when Lambda first receives the request and ends when the request leaves the Lambda service.
- **Lambda function (AWS::Lambda::Function)** – This type of node represents the Lambda function's execution time.
- **Downstream service calls** – In this type, each downstream service call from within the Lambda function is represented by a separate node.

In the diagram following, the nodes represent (from left to right): The Lambda service, the user function, and a downstream call to Amazon S3:



For more information, see [Viewing the Service Map](#).

Lambda as an AWS X-Ray Trace

From the service map, you can zoom in to see a trace view of your Lambda function. The trace will display in-depth information regarding your function invocations, represented as segments and subsegments:

- **Lambda service segment** – This segment represents different information depending on the event source used to invoke the function:
 - **Synchronous and stream event sources** – The service segment measures the time from when the Lambda service receives the request/event and ends when the request leaves the Lambda service (after the final invocation for the request is completed).
 - **Asynchronous** – The service segment represents the response time, that is, the time it took the Lambda service to return a 202 response to the client.

The Lambda service segment can include two types of subsegments:

- **Dwell time (asynchronous invocations only)** – Represents the time the function spends in the Lambda service before being invoked. This subsegment starts when the Lambda service receives the request/event and ends when the Lambda function is invoked for the first time.
- **Attempt** – Represents a single invocation attempt, including any overhead introduced by the Lambda service. Examples of overhead are time spent initializing the function's code and function execution time.
- **Lambda function segment** – Represents execution time for the function for a given invocation attempt. It starts when the function handler starts executing and ends when the function terminates. This segment can include three types of subsegments:
 - **Initialization** – The time spent running the initialization code of the function, defined as the code outside the Lambda function handler or static initializers.
 - **Downstream calls** – Calls made to other AWS services from the Lambda function's code.
 - **Custom subsegments** – Custom subsegments or user annotations that you can add to the Lambda function segment by using the X-Ray SDK.

Note

For each traced invocation, Lambda emits the Lambda service segment and all of its subsegments. These segments are emitted regardless of the runtime and require you to use the X-Ray SDK for AWS API calls.

Setting Up AWS X-Ray with Lambda

Following, you can find detailed information on how to set up X-Ray with Lambda.

Before You Begin

To enable tracing on your Lambda function using the Lambda CLI, you must first add tracing permissions to your function's execution role. To do so, take the following steps:

- Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
- Find the execution role for your Lambda function.
- Attach the following managed policy: `AWSXrayWriteOnlyAccess`

To learn more about these policies, see [AWS X-Ray](#).

If you are changing the tracing mode to active using the Lambda console, tracing permissions are added automatically, as explained in the next section.

Tracing

The path of a request through your application is tracked with a trace ID. A *trace* collects all of the segments generated by a single request, typically an HTTP GET or POST request.

There are two modes of tracing for a Lambda function:

- **Pass Through:** This is the default setting for all Lambda functions if you have added tracing permissions to your function's execution role. This approach means the Lambda function is only traced if X-Ray has been enabled on an upstream service, such as AWS Elastic Beanstalk.
- **Active:** When a Lambda function has this setting, Lambda automatically samples invocation requests, based on the sampling algorithm specified by X-Ray.

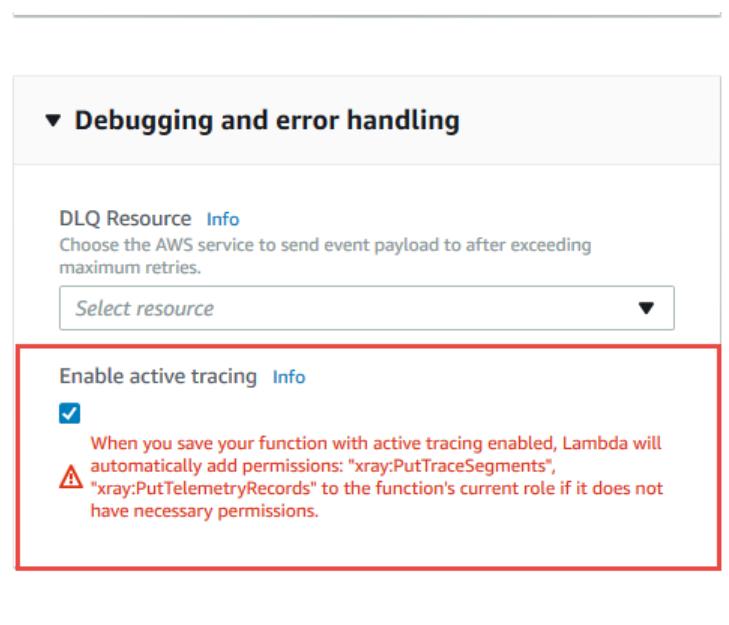
Note

X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of the requests that your application serves. The default sampling algorithm is 1 request per minute, with 5 percent of requests sampled past that limit. However, if the traffic volume to your function is low, you may see an increased rate of sampling.

You can change the tracing mode for your Lambda function by using either the Lambda Management Console or the Lambda [CreateFunction](#) (p. 355) or [UpdateFunctionConfiguration](#) (p. 463) API actions.

If you use the Lambda console, the following applies:

- When you change a function's tracing mode to active, tracing permissions are automatically attached to the function's execution role. If you receive an error stating Lambda couldn't add the `AWSXrayWriteOnlyAccess` policy to your function's execution role, sign in to the IAM console at <https://console.aws.amazon.com/iam/> and manually add the policy.
- To enable active tracing, go to the **Configuration** tab your function and select the **Enable active tracing** box.



If you use the Lambda [CreateFunction \(p. 355\)](#) or [UpdateFunctionConfiguration \(p. 463\)](#) API actions:

- If you want the tracing mode to be active, set the `TracingConfig` parameter's `Mode` property to `Active`. Again, any new function has its tracing mode set to `PassThrough` by default.
- Any new or updated Lambda function has its `$LATEST` version set to the value you specify.

Note

You receive an error if you haven't added tracing permissions to your function's execution role. For more information, see [Before You Begin \(p. 234\)](#).

Emitting Trace Segments from a Lambda Function

For each traced invocation, Lambda will emit the Lambda service segment and all of its subsegments. In addition, Lambda will emit the Lambda function segment and the init subsegment. These segments will be emitted regardless of the function's runtime, and with no code changes or additional libraries required. If you want your Lambda function's X-Ray traces to include custom segments, annotations, or subsegments for downstream calls, you might need to include additional libraries and annotate your code.

Note that any instrumentation code must be implemented inside the Lambda function handler and not as part of the initialization code.

The following examples explain how to do this in the supported runtimes:

- [Instrumenting Python Code in AWS Lambda \(p. 261\)](#)
- [Instrumenting Node.js Code in AWS Lambda \(p. 249\)](#)
- [Instrumenting Java Code in AWS Lambda \(p. 286\)](#)
- [???](#) (p. 300)

The AWS X-Ray Daemon in the Lambda Environment

The AWS X-Ray Daemon is a software application that gathers raw segment data and relays it to the AWS X-Ray service. The daemon works in conjunction with the AWS X-Ray SDKs so that data sent by the SDKs can reach the X-Ray service.

When you trace your Lambda function, the X-Ray daemon automatically runs in the Lambda environment to gather trace data and send it to X-Ray. When tracing, the X-Ray daemon consumes a maximum of 16 MB or 3 percent of your function's memory allocation. For example, if you allocate 128 MB of memory to your Lambda function, the X-Ray daemon has 16 MB of your function's memory allocation. If you allocate 1024 MB to your Lambda function, the X-Ray daemon has 31 MB allocated to it (3 percent). For more information, see [The AWS X-Ray Daemon](#).

Note

Lambda will try to terminate the X-Ray daemon to avoid exceeding your function's memory limit. For example, assume you have allocated 128 MB to your Lambda function, which means the X-Ray daemon will have 16 MB allocated to it. That leaves your Lambda function with a memory allocation of 112 MB. However, if your function exceeds 112 MB, the X-Ray daemon will be terminated to avoid throwing an out-of-memory error.

Using Environment Variables to Communicate with AWS X-Ray

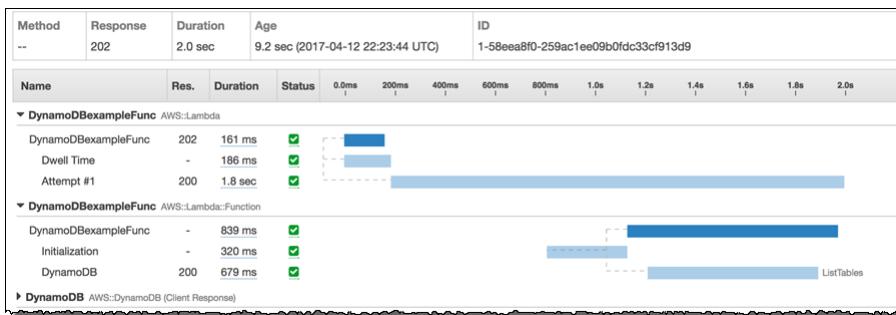
AWS Lambda uses environment variables to facilitate communication with the X-Ray daemon and configure the X-Ray SDK.

- **_X_AMZN_TRACE_ID:** Contains the tracing header, which includes the sampling decision, trace ID, and parent segment ID. (To learn more about these properties, see [Tracing Header](#).) If Lambda receives a tracing header when your function is invoked, that header will be used to populate the `_X_AMZN_TRACE_ID` environment variable. If a tracing header was not received, Lambda will generate one for you.
- **AWS_XRAY_CONTEXT_MISSING:** The X-Ray SDK uses this variable to determine its behavior in the event that your function tries to record X-Ray data, but a tracing header is not available. Lambda sets this value to `LOG_ERROR` by default.
- **AWS_XRAY_DAEMON_ADDRESS:** This environment variable exposes the X-Ray daemon's address in the following format: `IP_ADDRESS:PORT`. You can use the X-Ray daemon's address to send trace data to the X-Ray daemon directly, without using the X-Ray SDK.

Lambda Traces in the AWS X-Ray Console: Examples

The following shows Lambda traces for two different Lambda functions. Each trace showcases a trace structure for a different invocation type: asynchronous and synchronous.

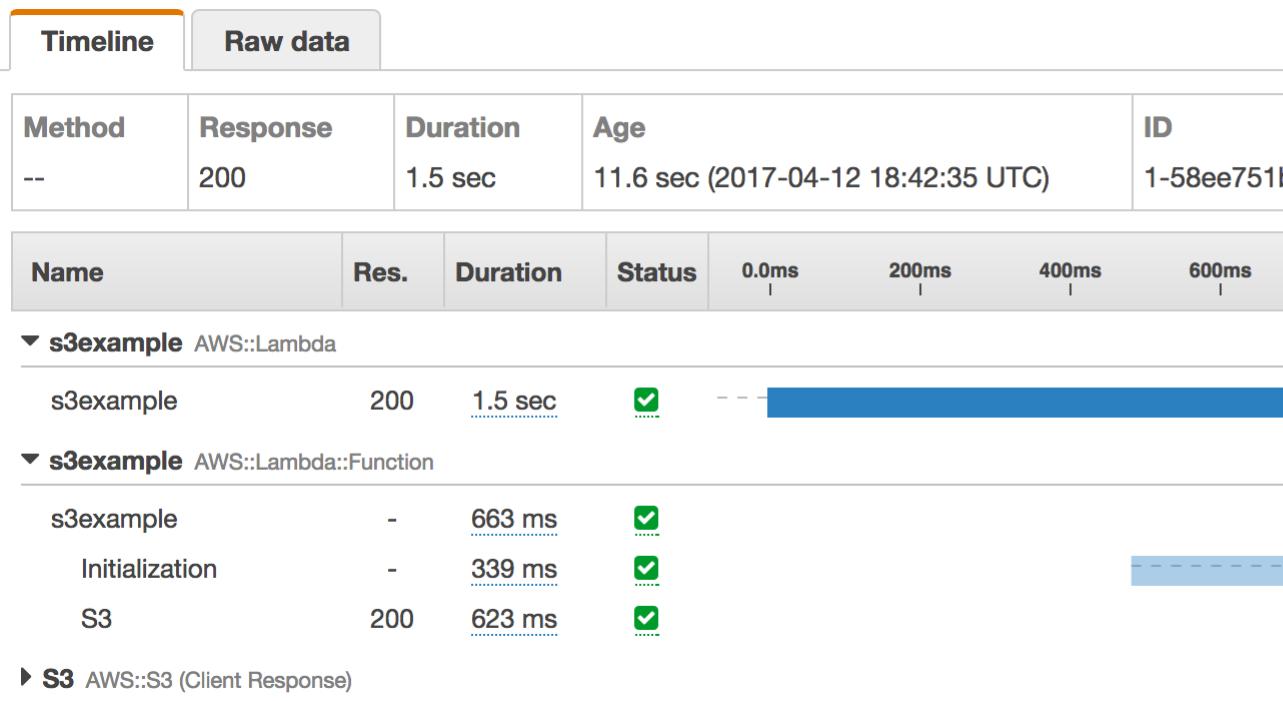
- **Async** – The example following shows an asynchronous Lambda request with one successful invocation and one downstream call to DynamoDB.



The Lambda service segment encapsulates the response time, which is the time it took to return a response (for example, 202) to the client. It includes subsegments for the time spent in the Lambda service queue (dwell time) and each invocation attempt. (Only one invocation attempt appears in the example preceding.) Each attempt subsegment in the service segment will have a corresponding user function segment. In this example, the user function segment contains two subsegments: the initialization subsegment representing the function's initialization code that is run before the handler, and a downstream call subsegment representing a `ListTables` call to DynamoDB.

Status codes and error messages are displayed for each Invocation subsegment and for each downstream call.

- **Synchronous** – The example following shows a synchronous request with one downstream call to Amazon S3.



The Lambda service segment captures the entire time the request spends in the Lambda service. The service segment will have a corresponding User function segment. In this example, the User function segment contains a subsegment representing the function's initialization code (code run before the handler), and a subsegment representing the `PutObject` call to Amazon S3.

Note

If you want to trace HTTP calls, you need to use an HTTP client. For more information, see [Tracing Calls to Downstream HTTP Web Services with the X-Ray SDK for Java](#) or [Tracing Calls to Downstream HTTP Web Services with the X-Ray SDK for Node.js](#).

Logging AWS Lambda API Calls with AWS CloudTrail

AWS Lambda is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in AWS Lambda. CloudTrail captures API calls for AWS Lambda as events. The calls captured include calls from the AWS Lambda console and code calls to the AWS Lambda API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for AWS Lambda. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to AWS Lambda, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, including how to configure and enable it, see the [AWS CloudTrail User Guide](#).

AWS Lambda Information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When supported event activity occurs in AWS Lambda, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see [Viewing Events with CloudTrail Event History](#).

For an ongoing record of events in your AWS account, including events for AWS Lambda, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- [Overview for Creating a Trail](#)
- [CloudTrail Supported Services and Integrations](#)
- [Configuring Amazon SNS Notifications for CloudTrail](#)
- [Receiving CloudTrail Log Files from Multiple Regions](#) and [Receiving CloudTrail Log Files from Multiple Accounts](#)

AWS Lambda supports logging the following actions as events in CloudTrail log files:

- [AddPermission \(p. 343\)](#)
- [CreateEventSourceMapping \(p. 351\)](#)
- [CreateFunction \(p. 355\)](#)

(The `ZipFile` parameter is omitted from the CloudTrail logs for `CreateFunction`.)
- [DeleteEventSourceMapping \(p. 365\)](#)
- [DeleteFunction \(p. 368\)](#)
- [GetEventSourceMapping \(p. 379\)](#)
- [GetFunction \(p. 382\)](#)

- [GetFunctionConfiguration \(p. 385\)](#)
- [GetPolicy \(p. 398\)](#)
- [ListEventSourceMappings \(p. 410\)](#)
- [ListFunctions \(p. 413\)](#)
- [RemovePermission \(p. 441\)](#)
- [UpdateEventSourceMapping \(p. 452\)](#)
- [UpdateFunctionCode \(p. 456\)](#)

(The `ZipFile` parameter is omitted from the CloudTrail logs for `UpdateFunctionCode`.)

- [UpdateFunctionConfiguration \(p. 463\)](#)

Every log entry contains information about who generated the request. The user identity information in the log helps you determine whether the request was made with root or IAM user credentials, with temporary security credentials for a role or federated user, or by another AWS service. For more information, see the `userIdentity` field in the [CloudTrail Event Reference](#).

You can store your log files in your bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted by using Amazon S3 server-side encryption (SSE).

You can choose to have CloudTrail publish Amazon SNS notifications when new log files are delivered if you want to take quick action upon log file delivery. For more information, see [Configuring Amazon SNS Notifications for CloudTrail](#).

You can also aggregate AWS Lambda log files from multiple AWS regions and multiple AWS accounts into a single S3 bucket. For more information, see [Working with CloudTrail Log Files](#).

Understanding AWS Lambda Log File Entries

CloudTrail log files contain one or more log entries where each entry is made up of multiple JSON-formatted events. A log entry represents a single request from any source and includes information about the requested action, any parameters, the date and time of the action, and so on. The log entries are not guaranteed to be in any particular order. That is, they are not an ordered stack trace of the public API calls.

The following example shows CloudTrail log entries for the `GetFunction` and `DeleteFunction` actions.

```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::999999999999:user/myUserName",
        "accountId": "999999999999",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:03:36Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "GetFunction",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "errorCode": "AccessDenied",
      "invocationType": "RequestResponse"
    }
  ]
}
```

```
    "errorMessage": "User: arn:aws:iam::999999999999:user/myUserName is not
authorized to perform: lambda:GetFunction on resource: arn:aws:lambda:us-
west-2:999999999999:function:other-acct-function",
    "requestParameters": null,
    "responseElements": null,
    "requestID": "7aebcd0f-cda1-11e4-aaa2-e356da31e4ff",
    "eventID": "e92a3e85-8ecd-4d23-8074-843aabfe89bf",
    "eventType": "AwsApiCall",
    "recipientAccountId": "999999999999"
},
{
    "eventVersion": "1.03",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::999999999999:user/myUserName",
        "accountId": "999999999999",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
    },
    "eventTime": "2015-03-18T19:04:42Z",
    "eventSource": "lambda.amazonaws.com",
    "eventName": "DeleteFunction",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "127.0.0.1",
    "userAgent": "Python-httplib2/0.8 (gzip)",
    "requestParameters": {
        "functionName": "basic-node-task"
    },
    "responseElements": null,
    "requestID": "a2198ecc-cda1-11e4-aaa2-e356da31e4ff",
    "eventID": "20b84ce5-730f-482e-b2b2-e8fcc87ceb22",
    "eventType": "AwsApiCall",
    "recipientAccountId": "999999999999"
}
]
```

Note

The `eventName` may include date and version information, such as `"GetFunction20150331"`, but it is still referring to the same public API. For more information, see [Services Supported by CloudTrail Event History](#) in the *AWS CloudTrail User Guide*.

Using CloudTrail to Track Function Invocations

CloudTrail also logs data events. You can turn on data event logging so that you log an event every time Lambda functions are invoked. This helps you understand what identities are invoking the functions and the frequency of their invocations. You can do this using the AWS CloudTrail console or [Invoke \(p. 400\)](#) CLI operation. For more information on this option, see [Logging Data and Management Events for Trails](#).

Building Lambda Functions with Node.js

AWS Lambda supports the following Node.js runtimes.

Node.js Runtimes

Name	Identifier	Node.js Version	AWS SDK for JavaScript	Operating System
Node.js 10	nodejs10.x	10.15	2.437.0	Amazon Linux 2
Node.js 8.10	nodejs8.10	8.10	2.290.0	Amazon Linux

When you create a Lambda function, you specify the runtime that you want to use. For more information, see [runtime parameter of CreateFunction \(p. 355\)](#).

The following sections explain how [common programming patterns and core concepts](#) apply when authoring Lambda function code in Node.js. The programming model described in the following sections applies to all supported runtime versions, except where indicated.

Topics

- [AWS Lambda Deployment Package in Node.js \(p. 241\)](#)
- [AWS Lambda Function Handler in Node.js \(p. 242\)](#)
- [AWS Lambda Context Object in Node.js \(p. 244\)](#)
- [AWS Lambda Function Logging in Node.js \(p. 245\)](#)
- [AWS Lambda Function Errors in Node.js \(p. 246\)](#)
- [Instrumenting Node.js Code in AWS Lambda \(p. 249\)](#)

AWS Lambda Deployment Package in Node.js

To create a Lambda function you first create a Lambda function deployment package, a .zip file consisting of your code and any dependencies.

You can create a deployment package yourself or write your code directly in the Lambda console, in which case the console creates the deployment package for you and uploads it, creating your Lambda function. Note the following to determine if you can use the console to create your Lambda function:

- **Simple scenario** – If your custom code requires only the AWS SDK library, then you can use the inline editor in the AWS Lambda console. Using the console, you can edit and upload your code to AWS Lambda. The console will zip up your code with the relevant configuration information into a deployment package that the Lambda service can run.

You can also test your code in the console by manually invoking it using sample event data.

Note

The Lambda service has preinstalled the AWS SDK for Node.js.

- **Advanced scenario** – If you are writing code that uses other resources, such as a graphics library for image processing, or you want to use the AWS CLI instead of the console, you need to first create the Lambda function deployment package, and then use the console or the CLI to upload the package.

Note

After you create a deployment package, you may either upload it directly or upload the .zip file first to an Amazon S3 bucket in the same AWS region where you want to create the Lambda function, and then specify the bucket name and object key name when you create the Lambda function using the console or the AWS CLI.

The following is an example procedure to create a deployment package (outside the console). Suppose you want to create a deployment package that includes a `filename.js` code file and your code uses the `async` library.

1. Open a text editor, and write your code. Save the file (for example, `filename.js`).

You will use the file name to specify the handler at the time of creating the Lambda function.

2. In the same directory, use `npm` to install the libraries that your code depends on. For example, if your code uses the `async` library, use the following `npm` command.

```
npm install async
```

3. Your directory will then have the following structure:

```
filename.js
node_modules/async
node_modules/async/lib
node_modules/async/lib/async.js
node_modules/async/package.json
```

4. Zip the content of the folder, that is your deployment package (for example, `sample.zip`).

Then, specify the .zip file name as your deployment package at the time you create your Lambda function.

If you want to include your own binaries, including native ones, just package them in the Zip file you upload and then reference them (including the relative path within the Zip file you created) when you call them from Node.js or from other processes that you've previously started. Ensure that you include the following at the start of your function code: `process.env['PATH'] = process.env['PATH'] + ':' + process.env['LAMBDA_TASK_ROOT']`

For more information on including native binaries in your Lambda function package, see [Running Executables in AWS Lambda](#). Also note that you need to supply the requisite permissions to the contents of the Zip file. For more information, see [Permissions Policies on Lambda Deployment Packages \(p. 32\)](#).

AWS Lambda Function Handler in Node.js

AWS Lambda invokes your Lambda function via a `handler` object. A `handler` represents the name of your Lambda function and serves as the entry point that AWS Lambda uses to execute your function code. For example:

```
exports.myHandler = function(event, context, callback) {
  ... function code
  callback(null, "some success message");
```

```
// or
// callback("some error type");
}
```

- **myHandler** – This is the name of the function AWS Lambda invokes. Suppose you save this code as `helloworld.js`. Then, `myHandler` is the function that contains your Lambda function code and `helloworld` is the name of the file that represents your deployment package. For more information, see [AWS Lambda Deployment Package in Node.js \(p. 241\)](#).
- **context** – AWS Lambda uses this parameter to provide details of your Lambda function's execution. For more information, see [AWS Lambda Context Object in Node.js \(p. 244\)](#).
- **callback (optional)** – See [Using the Callback Parameter \(p. 243\)](#).

Using the Callback Parameter

Node.js runtimes support the optional `callback` parameter. You can use it to explicitly return information back to the caller.

```
callback(Error error, Object result);
```

Both parameters are optional. `error` is an optional parameter that you can use to provide results of the failed Lambda function execution. When a Lambda function succeeds, you can pass `null` as the first parameter.

`result` is an optional parameter that you can use to provide the result of a successful function execution. The result provided must be `JSON.stringify` compatible. If an error is provided, this parameter is ignored.

If you don't use `callback` in your code, AWS Lambda will call it implicitly and the return value is `null`. When the callback is called, AWS Lambda continues the Lambda function invocation until the event loop is empty.

The following are example callbacks:

```
callback();      // Indicates success but no information returned to the caller.
callback(null); // Indicates success but no information returned to the caller.
callback(null, "success"); // Indicates success with information returned to the caller.
callback(error); // Indicates error with error information returned to the caller.
```

AWS Lambda treats any non-null value for the `error` parameter as a handled exception.

The `callback` method automatically logs the string representation of non-null values of `error` to the Amazon CloudWatch Logs stream associated with the Lambda function.

If the Lambda function was invoked synchronously, the `callback` returns a response body. If `error` is `null`, the response body is set to the string representation of `result`. If the `error` is not `null`, the `error` value will be populated in the response body.

When the `callback(error, null)` (and `callback(error)`) is called, Lambda will log the first 256 KB of the error object. For a larger error object, AWS Lambda truncates the log and displays the text `Truncated by Lambda` next to the error object.

If you are using runtime version 8.10, you can include the `async` keyword:

```
exports.myHandler = async function(event, context) {
    ...
}
```

```
// return information to the caller.  
}
```

Example

Consider the following example code.

```
exports.myHandler = function(event, context, callback) {  
    console.log("value1 = " + event.key1);  
    console.log("value2 = " + event.key2);  
    callback(null, "some success message");  
    // or  
    // callback("some error type");  
}
```

This example has one function, `myHandler`.

In the function, the `console.log()` statements log some of the incoming event data to CloudWatch Logs. When the `callback` parameter is called, the Lambda function exits only after the event loop passed is empty.

If you want to use the `async` feature provided by the v8.10 runtime, consider the following code sample:

```
exports.myHandler = async function(event, context) {  
    console.log("value1 = " + event.key1);  
    console.log("value2 = " + event.key2);  
    return "some success message";  
    // or  
    // throw new Error("some error type");  
}
```

AWS Lambda Context Object in Node.js

When Lambda runs your function, it passes a context object to the [handler \(p. 242\)](#). This object provides methods and properties that provide information about the invocation, function, and execution environment.

Context Methods

- `getRemainingTimeInMillis()` – Returns the number of milliseconds left before the execution times out.

Context Properties

- `functionName` – The name of the Lambda function.
- `functionVersion` – The [version \(p. 47\)](#) of the function.
- `invokedFunctionArn` – The Amazon Resource Name (ARN) used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memoryLimitInMB` – The amount of memory configured on the function.
- `awsRequestId` – The identifier of the invocation request.
- `logGroupName` – The log group for the function.

- `logStreamName` – The log stream for the function instance.
- `identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
 - `cognitoIdentityId` – The authenticated Amazon Cognito identity.
 - `cognitoIdentityPoolId` – The Amazon Cognito identity pool that authorized the invocation.
- `clientContext` – (mobile apps) Client context provided to the Lambda invoker by the client application.
 - `client.installation_id`
 - `client.app_title`
 - `client.app_version_name`
 - `client.app_version_code`
 - `client.app_package_name`
 - `env.platform_version`
 - `env.platform`
 - `env.make`
 - `env.model`
 - `env.locale`
 - `Custom` – Custom values set by the mobile application.
- `callbackWaitsForEmptyEventLoop` – Set to false to send the response right away when the [callback \(p. 243\)](#) executes, instead of waiting for the Node.js event loop to be empty. If false, any outstanding events will continue to run during the next invocation.

The following example shows a handler function that logs context information.

Example index.js

```
exports.handler = function(event, context, callback) {
    console.log('remaining time =', context.getRemainingTimeInMillis());
    console.log('functionName =', context.functionName);
    console.log('AWSrequestID =', context.awsRequestId);
    callback(null, context.functionName);
};
```

AWS Lambda Function Logging in Node.js

Your Lambda function comes with a CloudWatch Logs log group, with a log stream for each instance of your function. The runtime sends details about each invocation to the log stream, and relays logs and other output from your function's code.

To output logs from your function code, you can use methods on [the console object](#), or any logging library that writes to `stdout` or `stderr`. The following example logs the values of environment variables and the event object.

Example index.js

```
exports.handler = async (event) => {
    console.log("## ENVIRONMENT VARIABLES");
    console.log(JSON.stringify(process.env, null, 2));
    console.log("## EVENT");
    console.log(JSON.stringify(event, null, 2));
};
```

The Lambda console shows log output when you test a function on the function configuration page. To view logs for all invocations, use the CloudWatch Logs console.

To view your Lambda function's logs

1. Open the [Logs page of the CloudWatch console](#).
2. Choose the log group for your function (`/aws/lambda/function-name`).
3. Choose the first stream in the list.

Each log stream corresponds to an [instance of your function \(p. 104\)](#). New streams appear when you update your function and when additional instances are created to handle multiple concurrent invocations. To find logs for specific invocations, you can instrument your function with X-Ray and record details about the request and log stream in the trace. For a sample application that correlates logs and traces with X-Ray, see [Error Processor Sample Application for AWS Lambda \(p. 119\)](#).

To get logs for an invocation from the command line, use the `--log-type` option. The response includes a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

```
$ aws lambda invoke --function-name my-function out --log-type Tail
{
    "StatusCode": 200,
    "LogResult":
    "U1RBULQgUmVxdWVzdElkOia4N2QwNDRiOC1mMTU0LTEzTgtOGNkYS0yOTc0YzVlNGZimjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

You can use the `base64` utility to decode the logs.

```
$ aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text | base64 -d
START RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d Version: $LATEST
Processing event...
END RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d
REPORT RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d Duration: 29.40 ms      Billed
Duration: 100 ms          Memory Size: 128 MB      Max Memory Used: 19 MB
```

`base64` is available on Linux, macOS, and [Ubuntu on Windows](#). For macOS, the command is `base64 -D`.

Log groups are not deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.

AWS Lambda Function Errors in Node.js

If your Lambda function notifies AWS Lambda that it failed to execute properly, Lambda will attempt to convert the error object to a String. Consider the following example:

```
console.log('Loading function');

exports.handler = function(event, context, callback) {
    // This example code only throws error.
    var error = new Error("something is wrong");
    callback(error);
};
```

When you invoke this Lambda function, it will notify AWS Lambda that function execution completed with an error and passes the error information to AWS Lambda. AWS Lambda returns the error information back to the client:

```
{  
  "errorMessage": "something is wrong",  
  "errorType": "Error",  
  "stackTrace": [  
    "exports.handler (/var/task/index.js:10:17)"  
  ]  
}
```

You would get the same result if you write the function using the `async` feature of Node.js runtime version 8.10. For example:

```
exports.handler = async function(event, context) {  
  function AccountAlreadyExistsError(message) {  
    this.name = "AccountAlreadyExistsError";  
    this.message = message;  
  }  
  AccountAlreadyExistsError.prototype = new Error();  
  
  const error = new AccountAlreadyExistsError("Account is in use!");  
  throw error  
};
```

Again, when this Lambda function is invoked, it will notify AWS Lambda that function execution completed with an error and passes the error information to AWS Lambda. AWS Lambda returns the error information back to the client:

```
{  
  "errorMessage": "Account is in use!",  
  "errorType": "Error",  
  "stackTrace": [  
    "exports.handler (/var/task/index.js:10:17)"  
  ]  
}
```

Note that the error information is returned as the `stackTrace` JSON array of stack trace elements.

How you get the error information back depends on the invocation type that the client specifies at the time of function invocation:

- If a client specifies the `RequestResponse` invocation type (that is, synchronous execution), it returns the result to the client that made the `invoke` call.

For example, the console always use the `RequestResponse` invocation type, so the console will display the error in the **Execution result** section as shown:

✖ Execution result: failed (logs)

▼ Details

The area below shows the result returned by your function execution. [Learn more](#) about returning results from your function.

```
{  
  "errorMessage": "Something went wrong"  
}
```

The same information is also sent to CloudWatch and the **Log output** section shows the same logs.

Summary	Log output
Code SHA-256 U4b2T1IA6JHw7VXNt W02s3RXCtox6Ph3Jw3 7xQfO6g=	The area below shows the logging calls in your code. These correspond to a single row within the CloudWatch log group corresponding to this Lambda function. Click here to view the CloudWatch log group.
Request ID 093bf2a1-6cba-11e7-b9ad-850729ae85a8	START RequestId: 093bf2a1-6cba-11e7-b9ad-850729ae85a8 Version: \$LATEST 2017-07-19T19:39:53.469Z 093bf2a1-6cba-11e7-b9ad-850729ae85a8 value3 = undefined 2017-07-19T19:39:53.469Z 093bf2a1-6cba-11e7-b9ad-850729ae85a8 value4 = undefined 2017-07-19T19:39:53.488Z 093bf2a1-6cba-11e7-b9ad-850729ae85a8 value5 = undefined 2017-07-19T19:39:53.489Z 093bf2a1-6cba-11e7-b9ad-850729ae85a8 {"errorMessage": "Something went wrong"} END RequestId: 093bf2a1-6cba-11e7-b9ad-850729ae85a8 REPORT RequestId: 093bf2a1-6cba-11e7-b9ad-850729ae85a8 Duration: 41.24 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 17 MB
Duration 41.24 ms	
Billed duration 100 ms	

- If a client specifies the `Event` invocation type (that is, asynchronous execution), AWS Lambda will not return anything. Instead, it logs the error information to [CloudWatch Logs](#). You can also see the error metrics in [CloudWatch Metrics](#).

Depending on the event source, AWS Lambda may retry the failed Lambda function. For example, if Kinesis is the event source, AWS Lambda will retry the failed invocation until the Lambda function succeeds or the records in the stream expire. For more information on retries, see [AWS Lambda Retry Behavior \(p. 85\)](#).

To test the preceding Node.js code (console)

- In the console, create a Lambda function using the hello-world blueprint. In **runtime**, choose **Node.js** and, in **Role**, choose **Basic execution role**. For instructions on how to do this, see [Create a Lambda Function with the Console \(p. 3\)](#).
- Replace the template code with the code provided in this section.
- Test the Lambda function using the **Sample event template** called **Hello World** provided in the Lambda console.

Function Error Handling

You can create custom error handling to raise an exception directly from your Lambda function and handle it directly (Retry or Catch) within an AWS Step Functions State Machine. For more information, see [Handling Error Conditions Using a State Machine](#).

Consider a `CreateAccount` state is a **task** that writes a customer's details to a database using a Lambda function.

- If the task succeeds, an account is created and a welcome email is sent.
- If a user tries to create an account for a username that already exists, the Lambda function raises an error, causing the state machine to suggest a different username and to retry the account-creation process.

The following code samples demonstrate how to do this. Note that custom errors in Node.js must extend the `error` prototype.

```
exports.handler = function(event, context, callback) {
    function AccountAlreadyExistsError(message) {
        this.name = "AccountAlreadyExistsError";
        this.message = message;
    }
    AccountAlreadyExistsError.prototype = new Error();

    const error = new AccountAlreadyExistsError("Account is in use!");
    callback(error);
};
```

You can configure Step Functions to catch the error using a [Catch rule](#):

```
{
  "StartAt": "CreateAccount",
  "States": {
    "CreateAccount": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:CreateAccount",
      "Next": "SendWelcomeEmail",
      "Catch": [
        {
          "ErrorEquals": ["AccountAlreadyExistsError"],
          "Next": "SuggestAccountName"
        }
      ],
      ...
    }
  }
}
```

At runtime, AWS Step Functions catches the error, [transitioning](#) to the `SuggestAccountName` state as specified in the `Next` transition.

Note

The name property of the `Error` object must match the `ErrorEquals` value.

Custom error handling makes it easier to create [serverless](#) applications. This feature integrates with all the languages supported by the Lambda [Programming Model \(p. 31\)](#), allowing you to design your application in the programming languages of your choice, mixing and matching as you go.

To learn more about creating your own serverless applications using AWS Step Functions and AWS Lambda, see [AWS Step Functions](#).

Instrumenting Node.js Code in AWS Lambda

In Node.js, you can have Lambda emit subsegments to X-Ray to show you information about downstream calls to other AWS services made by your function. To do so, you first need to include the [the AWS X-Ray SDK for Node.js](#) in your deployment package. In addition, wrap your AWS SDK `require` statement in the following manner:

```
var AWSXRay = require('aws-xray-sdk-core');
var AWS = AWSXRay.captureAWS(require('aws-sdk'));
```

Then, use the `AWS` variable defined in the preceding example to initialize any service client that you want to trace with X-Ray, for example:

```
s3Client = AWS.S3();
```

After following these steps, any call made from your function using `s3Client` results in an X-Ray subsegment that represents that call. As an example, you can run the Node.js function following to see how the trace looks in X-Ray:

Example index.js

```
var AWSXRay = require('aws-xray-sdk-core');
var AWS = AWSXRay.captureAWS(require('aws-sdk'));

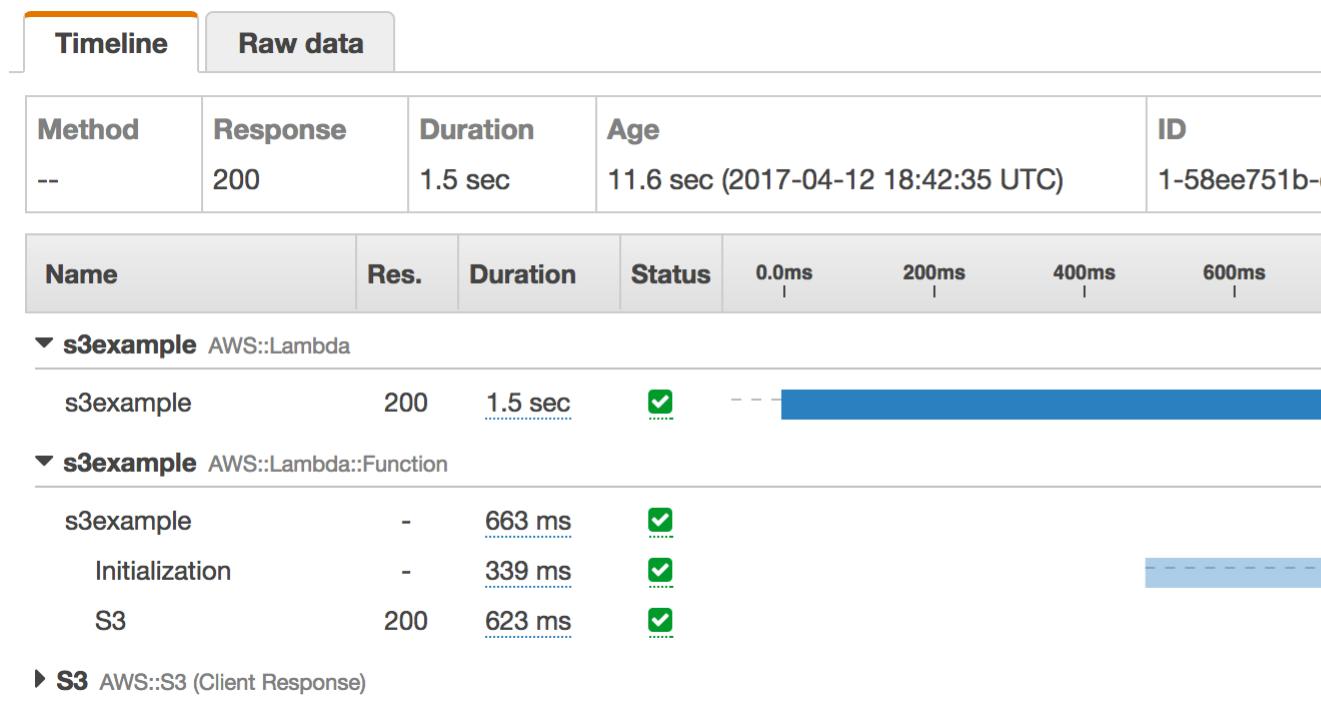
var s3 = new AWS.S3();

exports.handler = (event, context, callback) => {

    var params = {Bucket: process.env.BUCKET_NAME, Key: process.env.BUCKET_KEY, Body: process.env.BODY};

    s3.putObject(params, function(err, data) {
        if (err)
            { console.log(err) }
        else {
            console.log('success!')
        }
    });
};
```

Following is what a trace emitted by the code preceding looks like (asynchronous invocation):



Building Lambda Functions with Python

The following sections explain how [common programming patterns and core concepts](#) apply when authoring Lambda function code in Python.

Python Runtimes

Name	Identifier	AWS SDK for Python	Operating System
Python 3.6	python3.6	boto3-1.7.74 botocore-1.10.74	Amazon Linux
Python 3.7	python3.7	boto3-1.9.42 botocore-1.12.42	Amazon Linux
Python 2.7	python2.7	N/A	Amazon Linux

Topics

- [AWS Lambda Deployment Package in Python \(p. 251\)](#)
- [AWS Lambda Function Handler in Python \(p. 255\)](#)
- [AWS Lambda Context Object in Python \(p. 256\)](#)
- [AWS Lambda Function Logging in Python \(p. 257\)](#)
- [AWS Lambda Function Errors in Python \(p. 258\)](#)
- [Instrumenting Python Code in AWS Lambda \(p. 261\)](#)

AWS Lambda Deployment Package in Python

A deployment package is a ZIP archive that contains your function code and dependencies. You need to create a deployment package if you use the Lambda API to manage functions, or if your code uses libraries other than the AWS SDK. Other libraries and dependencies need to be included in the deployment package. You can upload the package directly to Lambda, or you can use an Amazon S3 bucket, and then upload it to Lambda.

If you use the Lambda [console editor \(p. 24\)](#) to author your function, the console manages the deployment package. You can use this method as long as you don't need to add any libraries. You can also use it to update a function that already has libraries in the deployment package, as long as the total size doesn't exceed 3 MB.

Note

You can use the AWS SAM CLI `build` command to create a deployment package for your Python function code and dependencies. See [Building Applications with Dependencies](#) in the AWS SAM Developer Guide for instructions.

Sections

- [Without Additional Dependencies \(p. 252\)](#)

- [With Additional Dependencies \(p. 252\)](#)
- [With a Virtual Environment \(p. 253\)](#)

Without Additional Dependencies

To create or update a function with the Lambda API, create an archive that contains your function code and upload it with the AWS CLI.

To update a Python function with no dependencies

1. Create a ZIP archive.

```
~/my-function$ zip function.zip function.py
```

2. Use the update-function-code command to upload the package.

```
~/my-function$ aws lambda update-function-code --function-name python37 --zip-file fileb://function.zip
{
    "FunctionName": "python37",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:python37",
    "Runtime": "python3.7",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Handler": "function.handler",
    "CodeSize": 815,
    "Description": "",
    "Timeout": 3,
    "MemorySize": 128,
    "LastModified": "2018-11-20T20:41:16.647+0000",
    "CodeSha256": "GcZ05oeHoJi61VpQj7vCLPs8DwCXmX5sE/fE2IHsizc=",
    "Version": "$LATEST",
    "VpcConfig": {
        "SubnetIds": [],
        "SecurityGroupIds": [],
        "VpcId": ""
    },
    "TracingConfig": {
        "Mode": "Active"
    },
    "RevisionId": "d1e983e3-ca8e-434b-8dc1-7add83d72ebd"
}
```

With Additional Dependencies

If your function depends on libraries other than the SDK for Python (Boto 3), install them to a local directory with [pip](#), and include them in your deployment package.

To update a Python function with dependencies

1. Create a directory for dependencies.

```
~/my-function$ mkdir package
```

2. Install libraries in the package directory with the --target option.

```
~/my-function$ cd package
~/my-function/package$ pip install Pillow --target .
```

```
Collecting Pillow
  Using cached https://files.pythonhosted.org/
  packages/62/8c/230204b8e968f6db00c765624f51cf1ecb6aea57b25ba00b240ee3fb0bd/
  Pillow-5.3.0-cp37-cp37m-manylinux1_x86_64.whl
  Installing collected packages: Pillow
  Successfully installed Pillow-5.3.0
```

3. Create a ZIP archive.

```
package$ zip -r9 ../function.zip .
  adding: PIL/ (stored 0%)
  adding: PIL/.libs/ (stored 0%)
  adding: PIL/.libs/libfreetype-7ce95de6.so.6.16.1 (deflated 65%)
  adding: PIL/.libs/libjpeg-3fe7dfc0.so.9.3.0 (deflated 72%)
  adding: PIL/.libs/liblcms2-a6801db4.so.2.0.8 (deflated 67%)
  ...
```

4. Add your function code to the archive.

```
~/my-function/package$ cd ../
~/my-function$ zip -g function.zip function.py
  adding: function.py (deflated 56%)
```

5. Update the function code.

```
~/my-function$ aws lambda update-function-code --function-name python37 --zip-file
fileb://function.zip
{
    "FunctionName": "python37",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:python37",
    "Runtime": "python3.7",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Handler": "function.handler",
    "CodeSize": 2269409,
    "Description": "",
    "Timeout": 3,
    "MemorySize": 128,
    "LastModified": "2018-11-20T20:51:35.871+0000",
    "CodeSha256": "GcZ05oeHoJi61VpQj7vCLPs8DwCXmX5sE/fE2IHsizc=",
    "Version": "$LATEST",
    "VpcConfig": {
        "SubnetIds": [],
        "SecurityGroupIds": [],
        "VpcId": ""
    },
    "TracingConfig": {
        "Mode": "Active"
    },
    "RevisionId": "a9c05ffd-8ad6-4d22-b6cd-d34a00c1702c"
}
```

With a Virtual Environment

In some cases, you may need to use a [virtual environment](#) to install dependencies for your function. This can occur if your function or its dependencies have dependencies on native libraries, or if you used Homebrew to install Python.

To update a Python function with a virtual environment

1. Create a virtual environment.

```
~/my-function$ virtualenv v-env
Using base prefix '/.local/python-3.7.0'
New python executable in v-env/bin/python3.7
Also creating executable in v-env/bin/python
Installing setuptools, pip, wheel...
done.
```

2. Activate the environment.

```
~/my-function$ source v-env/bin/activate
(v-env) ~/my-function$
```

For the Windows command line, the activation script is in the Scripts directory.

```
> v-env\Scripts\activate.bat
```

3. Install libraries with pip.

```
~/my-function$ pip install Pillow
Collecting Pillow
  Using cached https://files.pythonhosted.org/
packages/62/8c/230204b8e968f6db00c765624f51cf1ecb6aea57b25ba00b240ee3fb0bd/
Pillow-5.3.0-cp37-cp37m-manylinux1_x86_64.whl
Installing collected packages: Pillow
Successfully installed Pillow-5.3.0
```

4. Deactivate the virtual environment.

```
(v-env)~/my-function$ deactivate
```

5. Create a ZIP archive with the contents of the library.

```
~/my-function$ cd v-env/lib/python3.7/site-packages/
~/my-function/v-env/lib/python3.7/site-packages$ zip -r9 ../../../../function.zip .
  adding: easy_install.py (deflated 17%)
  adding: PIL/ (stored 0%)
  adding: PIL/.libs/ (stored 0%)
  adding: PIL/.libs/libfreetype-7ce95de6.so.6.16.1 (deflated 65%)
  adding: PIL/.libs/libjpeg-3fe7dfc0.so.9.3.0 (deflated 72%)
...
```

Depending on the library, dependencies may appear in either site-packages or dist-packages, and the first folder in the virtual environment may be lib or lib64. You can use the pip show command to locate a specific package.

6. Add your function code to the archive.

```
~/my-function/v-env/lib/python3.7/site-packages$ cd ../../..
~/my-function$ zip -g function.zip function.py
  adding: function.py (deflated 56%)
```

7. Update the function code.

```
~/my-function$ aws lambda update-function-code --function-name python37 --zip-file
fileb://function.zip
{
    "FunctionName": "python37",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:python37",
    "Runtime": "python3.7",
```

```
"Role": "arn:aws:iam::123456789012:role/lambda-role",
"Handler": "function.handler",
"CodeSize": 5912988,
"Description": "",
"Timeout": 3,
"MemorySize": 128,
"LastModified": "2018-11-20T21:08:26.326+0000",
"CodeSha256": "A2PONUWq1J+LsBkuP8tm9uNYqs1TAa3M76ptmZCw5g=",
"Version": "$LATEST",
"VpcConfig": {
    "SubnetIds": [],
    "SecurityGroupIds": [],
    "VpcId": ""
},
"TracingConfig": {
    "Mode": "Active"
},
"RevisionId": "5afdc7dc-2fcb-4ca8-8f24-947939ca707f"
}
```

AWS Lambda Function Handler in Python

At the time you create a Lambda function, you specify a *handler*, which is a function in your code, that AWS Lambda can invoke when the service executes your code. Use the following general syntax structure when creating a handler function in Python.

```
def handler_name(event, context):
    ...
    return some_value
```

In the syntax, note the following:

- **event** – AWS Lambda uses this parameter to pass in event data to the handler. This parameter is usually of the Python dict type. It can also be list, str, int, float, or NoneType type.
- **context** – AWS Lambda uses this parameter to provide runtime information to your handler. This parameter is of the LambdaContext type.
- Optionally, the handler can return a value. What happens to the returned value depends on the invocation type you use when invoking the Lambda function:
 - If you use the RequestResponse invocation type (synchronous execution), AWS Lambda returns the result of the Python function call to the client invoking the Lambda function (in the HTTP response to the invocation request, serialized into JSON). For example, AWS Lambda console uses the RequestResponse invocation type, so when you invoke the function using the console, the console will display the returned value.

If the handler returns `NONE`, AWS Lambda returns null.

- If you use the Event invocation type (asynchronous execution), the value is discarded.

For example, consider the following Python example code.

```
def my_handler(event, context):
    message = 'Hello {} {}!'.format(event['first_name'],
                                    event['last_name'])
    return {
        'message' : message
    }
```

This example has one function called `my_handler`. The function returns a message containing data from the event it received as input.

AWS Lambda Context Object in Python

When Lambda runs your function, it passes a context object to the [handler \(p. 255\)](#). This object provides methods and properties that provide information about the invocation, function, and execution environment.

Context Methods

- `get_remaining_time_in_millis` – Returns the number of milliseconds left before the execution times out.

Context Properties

- `function_name` – The name of the Lambda function.
- `function_version` – The [version \(p. 47\)](#) of the function.
- `invoked_function_arn` – The Amazon Resource Name (ARN) used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memory_limit_in_mb` – The amount of memory configured on the function.
- `aws_request_id` – The identifier of the invocation request.
- `log_group_name` – The log group for the function.
- `log_stream_name` – The log stream for the function instance.
- `identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
 - `cognito_identity_id` – The authenticated Amazon Cognito identity.
 - `cognito_identity_pool_id` – The Amazon Cognito identity pool that authorized the invocation.
- `client_context` – (mobile apps) Client context provided to the Lambda invoker by the client application.
 - `client.installation_id`
 - `client.app_title`
 - `client.app_version_name`
 - `client.app_version_code`
 - `client.app_package_name`
 - `custom` – A dict of custom values set by the mobile client application.
 - `env` – A dict of environment information provided by the AWS SDK.

The following example shows a handler function that logs context information.

Example `handler.py`

```
import time
def get_my_log_stream(event, context):
    print("Log stream name:", context.log_stream_name)
    print("Log group name:", context.log_group_name)
    print("Request ID:", context.aws_request_id)
    print("Mem. limits(MB):", context.memory_limit_in_mb)
    # Code will execute quickly, so we add a 1 second intentional delay so you can see that
    # in time remaining value.
    time.sleep(1)
```

```
print("Time remaining (MS):", context.get_remaining_time_in_millis())
```

In addition to the options listed above, you can also use the AWS X-Ray SDK for [Instrumenting Python Code in AWS Lambda \(p. 261\)](#) to identify critical code paths, trace their performance and capture the data for analysis.

AWS Lambda Function Logging in Python

Your Lambda function comes with a CloudWatch Logs log group, with a log stream for each instance of your function. The runtime sends details about each invocation to the log stream, and relays logs and other output from your function's code.

To output logs from your function code, you can use the [print method](#), or any logging library that writes to `stdout` or `stderr`. The following example logs the values of environment variables and the event object.

Example `lambda_function.py`

```
import json
import os

def lambda_handler(event, context):
    print('## ENVIRONMENT VARIABLES')
    print(os.environ)
    print('## EVENT')
    print(event)
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

For more detailed logs, use the [logging library](#).

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

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES')
    logger.info(os.environ)
    logger.info('## EVENT')
    logger.info(event)
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

The output from `logger` includes the log level, timestamp, and request ID.

```
[INFO] 2019-04-21T23:24:14.135Z 00d3cdad-8aaf-42b2-af4e-6f8b2cae00a5 ## EVENT
[INFO] 2019-04-21T23:24:14.135Z 00d3cdad-8aaf-42b2-af4e-6f8b2cae00a5 {'key1': 'value1',
'key2': 'value2', 'key3': 'value3'}
```

The Lambda console shows log output when you test a function on the function configuration page. To view logs for all invocations, use the CloudWatch Logs console.

To view your Lambda function's logs

1. Open the [Logs page of the CloudWatch console](#).
2. Choose the log group for your function (`/aws/lambda/function-name`).
3. Choose the first stream in the list.

Each log stream corresponds to an [instance of your function \(p. 104\)](#). New streams appear when you update your function and when additional instances are created to handle multiple concurrent invocations. To find logs for specific invocations, you can instrument your function with X-Ray and record details about the request and log stream in the trace. For a sample application that correlates logs and traces with X-Ray, see [Error Processor Sample Application for AWS Lambda \(p. 119\)](#).

To get logs for an invocation from the command line, use the `--log-type` option. The response includes a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

```
$ aws lambda invoke --function-name my-function out --log-type Tail
{
    "StatusCode": 200,
    "LogResult":
    "U1RBULQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTexZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

You can use the `base64` utility to decode the logs.

```
$ aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text | base64 -d
START RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d Version: $LATEST
Processing event...
END RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d
REPORT RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d Duration: 29.40 ms      Billed
Duration: 100 ms          Memory Size: 128 MB      Max Memory Used: 19 MB
```

`base64` is available on Linux, macOS, and [Ubuntu on Windows](#). For macOS, the command is `base64 -D`.

Log groups are not deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.

AWS Lambda Function Errors in Python

If your Lambda function raises an exception, AWS Lambda recognizes the failure and serializes the exception information into JSON and returns it. Consider the following example:

```
def always_failed_handler(event, context):
    raise Exception('I failed!')
```

When you invoke this Lambda function, it will raise an exception and AWS Lambda returns the following error message:

```
{
    "errorMessage": "I failed!",
    "stackTrace": [
        "/var/task/lambda_function.py",
        3,
        "my_always_fails_handler",
```

```

        "raise Exception('I failed!')"
    ],
],
"errorType": "Exception"
}

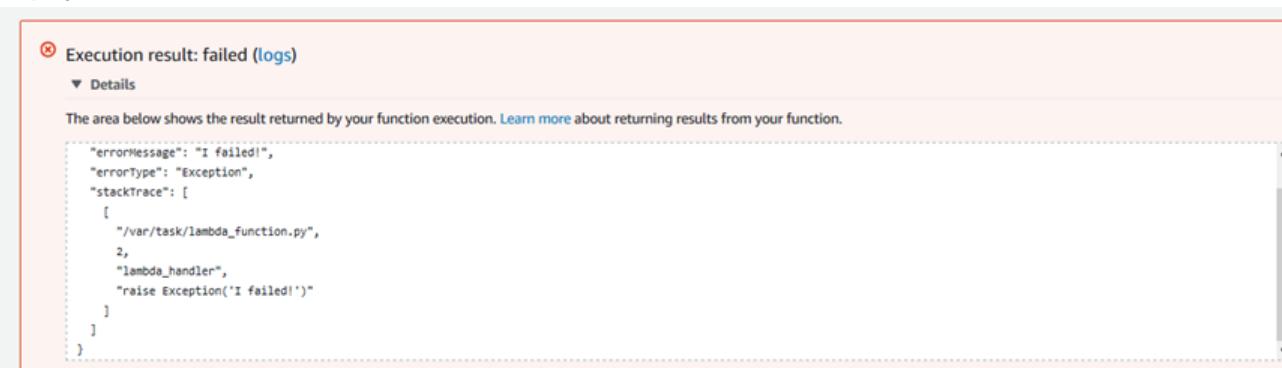
```

Note that the stack trace is returned as the `stackTrace` JSON array of stack trace elements.

How you get the error information back depends on the invocation type that the client specifies at the time of function invocation:

- If a client specifies the `RequestResponse` invocation type (that is, synchronous execution), it returns the result to the client that made the invoke call.

For example, the console always use the `RequestResponse` invocation type, so the console will display the error in the **Execution result** section as shown:



The screenshot shows the "Execution result" section of the AWS Lambda console for a failed function execution. The title is "Execution result: failed (logs)". Below it is a "Details" section with a note about returning results from your function. The main content area displays the error message and stack trace in JSON format:

```

{
  "errorMessage": "I failed!",
  "errorType": "Exception",
  "stackTrace": [
    [
      "/var/task/lambda_function.py",
      2,
      "lambda_handler",
      "raise Exception('I failed!')"
    ]
  ]
}

```

The same information is also sent to CloudWatch and the **Log output** section shows the same logs.



The screenshot shows the "Log output" section of the AWS Lambda console. It includes a "Summary" table and a log message. The summary table shows:

Code SHA-256	Qff84kaKBoCUM7FYFsoZNnnsDTQvF9MpVmRScWAFWvk=	Request ID	b5f4b0df-d47a-11e7-bac5-d145e32e9b46
Duration	0.82 ms	Billed duration	100 ms
Resources configured	128 MB	Max memory used	22 MB

The "Log output" section shows the following log message:

```

START RequestId: b5f4b0df-d47a-11e7-bac5-d145e32e9b46 Version: $LATEST
I failed: Exception
Traceback (most recent call last):
  File "/var/task/lambda_function.py", line 2, in lambda_handler
    raise Exception('I failed!')
Exception: I failed!

END RequestId: b5f4b0df-d47a-11e7-bac5-d145e32e9b46
REPORT RequestId: b5f4b0df-d47a-11e7-bac5-d145e32e9b46 Duration: 0.82 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 22 MB

```

- If a client specifies the `Event` invocation type (that is, asynchronous execution), AWS Lambda will not return anything. Instead, it logs the error information to CloudWatch Logs. You can also see the error metrics in CloudWatch Metrics.

Depending on the event source, AWS Lambda may retry the failed Lambda function. For example, if Kinesis is the event source, AWS Lambda will retry the failed invocation until the Lambda function succeeds or the records in the stream expire.

To test the preceding Python code (console)

1. In the console, create a Lambda function using the hello-world blueprint. In **runtime**, choose Python 3.7. In **Handler**, replace `lambda_function.lambda_handler` with

`lambda_function.always_failed_handler`. For instructions on how to do this, see [Create a Lambda Function with the Console \(p. 3\)](#).

2. Replace the template code with the code provided in this section.
3. Test the Lambda function using the **Sample event template** called **Hello World** provided in the Lambda console.

Function Error Handling

You can create custom error handling to raise an exception directly from your Lambda function and handle it directly (Retry or Catch) within an AWS Step Functions State Machine. For more information, see [Handling Error Conditions Using a State Machine](#).

Consider a `CreateAccount` state is a **task** that writes a customer's details to a database using a Lambda function.

- If the task succeeds, an account is created and a welcome email is sent.
- If a user tries to create an account for a username that already exists, the Lambda function raises an error, causing the state machine to suggest a different username and to retry the account-creation process.

The following code samples demonstrate how to do this. Note that custom errors in Python must extend the `Exception` class.

```
class AccountAlreadyExistsException(Exception): pass

def create_account(event, context):
    raise AccountAlreadyExistsException('Account is in use!')
```

You can configure Step Functions to catch the error using a `Catch` rule. Lambda automatically sets the error name to the simple class name of the exception at runtime:

```
{
  "StartAt": "CreateAccount",
  "States": {
    "CreateAccount": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:CreateAccount",
      "Next": "SendWelcomeEmail",
      "Catch": [
        {
          "ErrorEquals": [ "AccountAlreadyExistsException" ],
          "Next": "SuggestAccountName"
        }
      ]
    },
    ...
  }
}
```

At runtime, AWS Step Functions catches the error, [transitioning](#) to the `SuggestAccountName` state as specified in the `Next` transition.

Custom error handling makes it easier to create [serverless](#) applications. This feature integrates with all the languages supported by the Lambda [Programming Model \(p. 31\)](#), allowing you to design your application in the programming languages of your choice, mixing and matching as you go.

To learn more about creating your own serverless applications using AWS Step Functions and AWS Lambda, see [AWS Step Functions](#).

Instrumenting Python Code in AWS Lambda

In Python, you can have Lambda emit subsegments to X-Ray to show you information about downstream calls to other AWS services made by your function. To do so, you first need to include the [the AWS X-Ray SDK for Python](#) in your deployment package. In addition, you can patch the `boto3` (or `botocore` if you are using sessions), so any client you create to access other AWS services will automatically be traced by X-Ray.

```
import boto3
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch

patch(['boto3'])
```

Once you've patched the module you are using to create clients, you can use it to create your traced clients, in the case below Amazon S3:

```
s3_client = boto3.client('s3')
```

The X-Ray SDK for Python creates a subsegment for the call and records information from the request and response. You can use the `aws_xray_sdk.core.xray_recorder` to create subsegments automatically by decorating your Lambda functions or manually by calling `xray_recorder.begin_subsegment()` and `xray_recorder.end_subsegment()` inside the function, as shown in the following Lambda function.

```
import boto3
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch

patch(['boto3'])

s3_client = boto3.client('s3')

def lambda_handler(event, context):
    bucket_name = event['bucket_name']
    bucket_key = event['bucket_key']
    body = event['body']

    put_object_into_s3(bucket_name, bucket_key, body)
    get_object_from_s3(bucket_name, bucket_key)

# Define subsegments manually
def put_object_into_s3(bucket_name, bucket_key, body):
    try:
        xray_recorder.begin_subsegment('put_object')
        response = s3_client.put_object(Bucket=bucket_name, Key=bucket_key, Body=body)
        status_code = response['ResponseMetadata']['HTTPStatusCode']
        xray_recorder.current_subsegment().put_annotation('put_response', status_code)
    finally:
        xray_recorder.end_subsegment()

# Use decorators to automatically set the subsegments
@xray_recorder.capture('get_object')
def get_object_from_s3(bucket_name, bucket_key):
    response = s3_client.get_object(Bucket=bucket_name, Key=bucket_key)
```

```
status_code = response['ResponseMetadata']['HTTPStatusCode']
xray_recorder.current_subsegment().put_annotation('get_response', status_code)
```

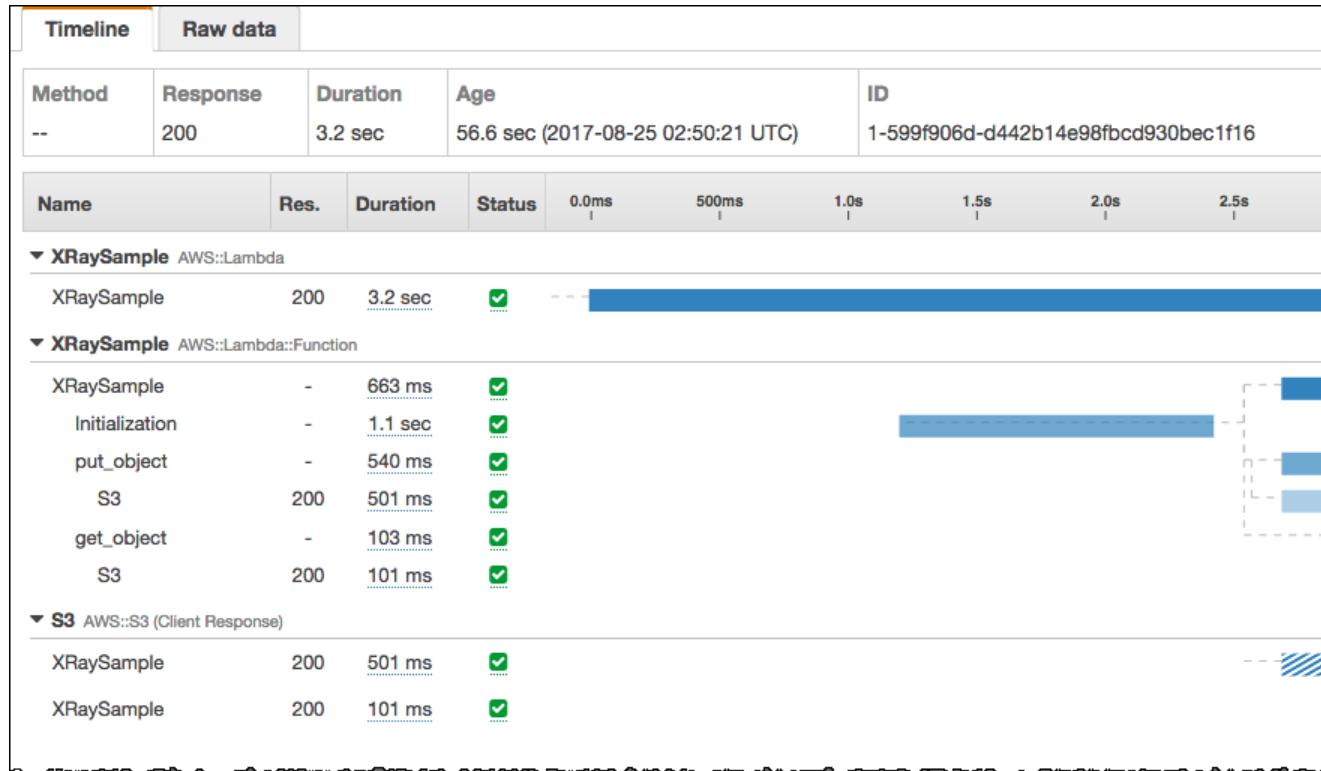
Note

The X-Ray SDK for Python allows you to patch the following modules:

- botocore
- boto3
- requests
- sqlite3
- mysql

You can use `patch_all()` to patch all of them at once.

Following is what a trace emitted by the code preceding looks like (synchronous invocation):



Building Lambda Functions with Java

The following sections explain how [common programming patterns and core concepts](#) apply when authoring Lambda function code in Java.

Java Runtimes

Name	Identifier	JDK	Operating System
Java 8	java8	java-1.8.0-openjdk	Amazon Linux

Topics

- [AWS Lambda Deployment Package in Java \(p. 264\)](#)
- [AWS Lambda Function Handler in Java \(p. 271\)](#)
- [AWS Lambda Context Object in Java \(p. 281\)](#)
- [AWS Lambda Function Logging in Java \(p. 281\)](#)
- [AWS Lambda Function Errors in Java \(p. 284\)](#)
- [Instrumenting Java Code in AWS Lambda \(p. 286\)](#)
- [Create a Lambda Function Authored in Java \(p. 288\)](#)

Additionally, note that AWS Lambda provides the following libraries:

- **aws-lambda-java-core** – This library provides the Context object, RequestStreamHandler, and the RequestHandler interfaces. The Context object ([AWS Lambda Context Object in Java \(p. 281\)](#)) provides runtime information about your Lambda function. The predefined interfaces provide one way of defining your Lambda function handler. For more information, see [Leveraging Predefined Interfaces for Creating Handler \(Java\) \(p. 277\)](#).
- **aws-lambda-java-events** – This library provides predefined types that you can use when writing Lambda functions to process events published by Amazon S3, Kinesis, Amazon SNS, and Amazon Cognito. These classes help you process the event without having to write your own custom serialization logic.
- **Custom Appender for Log4j2.8** – You can use the custom Log4j (see [Apache Log4j 2](#)) appender provided by AWS Lambda for logging from your lambda functions. Every call to Log4j methods, such as log.info() or log.error(), will result in a CloudWatch Logs event. The custom appender is called LambdaAppender and must be used in the log4j2.xml file. You must include the aws-lambda-java-log4j2 artifact (artifactId:aws-lambda-java-log4j2) in the deployment package (.jar file). For more information, see [AWS Lambda Function Logging in Java \(p. 281\)](#).
- **Custom Appender for Log4j1.2** – You can use the custom Log4j (see [Apache Log4j 1.2](#)) appender provided by AWS Lambda for logging from your lambda functions. For more information, see [AWS Lambda Function Logging in Java \(p. 281\)](#).

Note

Support for the Log4j v1.2 custom appender is marked for End-Of-Life. It will not receive ongoing updates and is not recommended for use.

These libraries are available through the [Maven Central Repository](#) and can also be found on [GitHub](#).

AWS Lambda Deployment Package in Java

Your deployment package can be a .zip file or a standalone jar; it is your choice. You can use any build and packaging tool you are familiar with to create a deployment package.

We provide examples of using Maven to create standalone jars and using Gradle to create a .zip file. For more information, see the following topics:

Topics

- [Creating a .jar Deployment Package Using Maven without any IDE \(Java\) \(p. 264\)](#)
- [Creating a .jar Deployment Package Using Maven and Eclipse IDE \(Java\) \(p. 266\)](#)
- [Creating a ZIP Deployment Package for a Java Function \(p. 268\)](#)
- [Authoring Lambda Functions Using Eclipse IDE and AWS SDK Plugin \(Java\) \(p. 271\)](#)

Creating a .jar Deployment Package Using Maven without any IDE (Java)

This section shows how to package your Java code into a deployment package using Maven at the command line.

Topics

- [Before You Begin \(p. 264\)](#)
- [Project Structure Overview \(p. 264\)](#)
- [Step 1: Create Project \(p. 265\)](#)
- [Step 2: Build Project \(Create Deployment Package\) \(p. 266\)](#)

Before You Begin

You will need to install the Maven command-line build tool. For more information, go to [Maven](#). If you are using Linux, check your package manager.

```
sudo apt-get install mvn
```

if you are using Homebrew

```
brew install maven
```

Project Structure Overview

After you set up the project, you should have the following folder structure:

```
project-dir/pom.xml  
project-dir/src/main/java/ (your code goes here)
```

Your code will then be in the /java folder. For example, if your package name is `example` and you have a `Hello.java` class in it, the structure will be:

```
project-dir/src/main/java/example/Hello.java
```

After you build the project, the resulting .jar file (that is, your deployment package), will be in the `project-dir/target` subdirectory.

Step 1: Create Project

Follow the steps in this section to create a Java project.

1. Create a project directory (`project-dir`).
2. In the `project-dir` directory, create the following:
 - Project Object Model file, `pom.xml`. Add the following project information and configuration details for Maven to build the project.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>doc-examples</groupId>
  <artifactId>lambda-java-example</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>lambda-java-example</name>

  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.1.0</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.3</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Note

- In the `dependencies` section, the `groupId` (that is, `com.amazonaws`) is the Amazon AWS group ID for Maven artifacts in the Maven Central Repository. The `artifactId` (that is, `aws-lambda-java-core`) is the AWS Lambda core library that provides definitions

of the `RequestHandler`, `RequestStreamHandler`, and the `Context` AWS Lambda interfaces for use in your Java application. At the build time Maven resolves these dependencies.

- In the plugins section, the Apache `maven-shade-plugin` is a plugin that Maven will download and use during your build process. This plugin is used for packaging jars to create a standalone .jar (a .zip file), your deployment package.
- If you are following other tutorial topics in this guide, the specific tutorials might require you to add more dependencies. Make sure to add those dependencies as required.

3. In the `project-dir`, create the following structure:

```
project-dir/src/main/java
```

4. Under the `/java` subdirectory you add your Java files and folder structure, if any. For example, if your Java package name is `example`, and source code is `Hello.java`, your directory structure looks like this:

```
project-dir/src/main/java/example/Hello.java
```

Step 2: Build Project (Create Deployment Package)

Now you can build the project using Maven at the command line.

1. At a command prompt, change directory to the project directory (`project-dir`).
2. Run the following `mvn` command to build the project:

```
$ mvn package
```

The resulting .jar is saved as `project-dir/target/lambda-java-example-1.0-SNAPSHOT.jar`. The .jar name is created by concatenating the `artifactId` and `version` in the `pom.xml` file.

The build creates this resulting .jar, using information in the `pom.xml` to do the necessary transforms. This is a standalone .jar (.zip file) that includes all the dependencies. This is your deployment package that you can upload to AWS Lambda to create a Lambda function.

Creating a .jar Deployment Package Using Maven and Eclipse IDE (Java)

This section shows how to package your Java code into a deployment package using Eclipse IDE and Maven plugin for Eclipse.

Topics

- [Before You Begin \(p. 266\)](#)
- [Step 1: Create and Build a Project \(p. 267\)](#)

Before You Begin

Install the **Maven** Plugin for Eclipse.

1. Start Eclipse. From the **Help** menu in Eclipse, choose **Install New Software**.
2. In the **Install** window, type `http://download.eclipse.org/technology/m2e/releases` in the **Work with:** box, and choose **Add**.
3. Follow the steps to complete the setup.

Step 1: Create and Build a Project

In this step, you start Eclipse and create a Maven project. You will add the necessary dependencies, and build the project. The build will produce a .jar, which is your deployment package.

1. Create a new Maven project in Eclipse.
 - a. From the **File** menu, choose **New**, and then choose **Project**.
 - b. In the **New Project** window, choose **Maven Project**.
 - c. In the **New Maven Project** window, choose **Create a simple project**, and leave other default selections.
 - d. In the **New Maven Project, Configure project** windows, type the following **Artifact** information:
 - **Group Id:** doc-examples
 - **Artifact Id:** lambda-java-example
 - **Version:** 0.0.1-SNAPSHOT
 - **Packaging:** jar
 - **Name:** lambda-java-example
2. Add the `aws-lambda-java-core` dependency to the `pom.xml` file.

It provides definitions of the `RequestHandler`, `RequestStreamHandler`, and `Context` interfaces. This allows you to compile code that you can use with AWS Lambda.

- a. Open the context (right-click) menu for the `pom.xml` file, choose **Maven**, and then choose **Add Dependency**.
- b. In the **Add Dependency** windows, type the following values:

Group Id: com.amazonaws

Artifact Id: aws-lambda-java-core

Version: 1.1.0

Note

If you are following other tutorial topics in this guide, the specific tutorials might require you to add more dependencies. Make sure to add those dependencies as required.

3. Add Java class to the project.
 - a. Open the context (right-click) menu for the `src/main/java` subdirectory in the project, choose **New**, and then choose **Class**.
 - b. In the **New Java Class** window, type the following values:
 - **Package:** example
 - **Name:** Hello

Note

If you are following other tutorial topics in this guide, the specific tutorials might recommend different package name or class name.

- c. Add your Java code. If you are following other tutorial topics in this guide, add the provided code.
4. Build the project.

Open the context (right-click) menu for the project in **Package Explorer**, choose **Run As**, and then choose **Maven Build** In the **Edit Configuration** window, type **package** in the **Goals** box.

Note

The resulting .jar, `lambda-java-example-0.0.1-SNAPSHOT.jar`, is not the final standalone .jar that you can use as your deployment package. In the next step, you add the Apache `maven-shade-plugin` to create the standalone .jar. For more information, go to [Apache Maven Shade Plugin](#).

5. Add the `maven-shade-plugin` plugin and rebuild.

The `maven-shade-plugin` will take artifacts (jars) produced by the *package* goal (produces customer code .jar), and create a standalone .jar that contains the compiled customer code, and the resolved dependencies from the `pom.xml`.

- a. Open the context (right-click) menu for the `pom.xml` file, choose **Maven**, and then choose **Add Plugin**.
- b. In the **Add Plugin** window, type the following values:
 - **Group Id:** org.apache.maven.plugins
 - **Artifact Id:** maven-shade-plugin
 - **Version:** 2.3
- c. Now build again.

This time we will create the jar as before, and then use the `maven-shade-plugin` to pull in dependencies to make the standalone .jar.

- i. Open the context (right-click) menu for the project, choose **Run As**, and then choose **Maven build**
- ii. In the **Edit Configuration** windows, type **package shade:shade** in the **Goals** box.
- iii. Choose **Run**.

You can find the resulting standalone .jar (that is, your deployment package), in the / `target` subdirectory.

Open the context (right-click) menu for the /`target` subdirectory, choose **Show In**, choose **System Explorer**, and you will find the `lambda-java-example-0.0.1-SNAPSHOT.jar`.

Creating a ZIP Deployment Package for a Java Function

This section provides examples of creating .zip file as your deployment package. You can use any build and packaging tool you like to create a deployment package with the following structure.

- All compiled class files and resource files at the root level.
- All required jars to run the code in the /`lib` directory.

Note

Lambda loads JARs in unicode alphabetical order. If multiple JARs in the `lib` folder contain the same class, the first one is used. You can use the following shell script to identify duplicate classes.

Example test-zip.sh

```
mkdir -p expanded
unzip path/to/my/function.zip -d expanded
find ./expanded/lib -name '*.jar' | xargs -n1 zipinfo -1 | grep '.*.class' | sort
| uniq -c | sort
```

The following examples use Gradle build and deployment tool to create a deployment package.

Before You Begin

You will need to download Gradle version 2.0 or later. For instructions, go to the gradle website, <https://gradle.org/>.

Example 1: Creating .zip Using Gradle and the Maven Central Repository

At the end of this walkthrough, you will have a project directory (*project-dir*) with content having the following structure:

```
project-dir/build.gradle
project-dir/src/main/java/
```

The */java* folder will contain your code. For example, if your package name is `example`, and you have a `Hello.java` class in it, the structure will be:

```
project-dir/src/main/java/example/Hello.java
```

After you build the project, the resulting .zip file (that is, your deployment package), will be in the *project-dir/build/distributions* subdirectory.

1. Create a project directory (*project-dir*).
2. In the *project-dir*, create `build.gradle` file and add the following content:

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile (
        'com.amazonaws:aws-lambda-java-core:1.1.0',
        'com.amazonaws:aws-lambda-java-events:1.1.0'
    )
}

task buildZip(type: Zip) {
    from compileJava
    from processResources
    into('lib') {
        from configurations.compileClasspath
    }
}
build.dependsOn buildZip
```

Note

- The repositories section refers to Maven Central Repository. At build time, it fetches the dependencies (that is, the two AWS Lambda libraries) from Maven Central.
- The `buildZip` task describes how to create the deployment package .zip file.

For example, if you unzip the resulting .zip file you should find any of the compiled class files and resource files at the root level. You should also find a /lib directory with the required jars for running the code.

- If you are following other tutorial topics in this guide, the specific tutorials might require you to add more dependencies. Make sure to add those dependencies as required.

3. In the `project-dir`, create the following structure:

```
project-dir/src/main/java/
```

4. Under the /java subdirectory you add your Java files and folder structure, if any. For example, if you Java package name is `example`, and source code is `Hello.java`, then your directory structure looks like this:

```
project-dir/src/main/java/example/Hello.java
```

5. Run the following gradle command to build and package the project in a .zip file.

```
project-dir> gradle build
```

6. Verify the resulting `project-dir.zip` file in the `project-dir/build/distributions` subdirectory.
7. Now you can upload the .zip file, your deployment package to AWS Lambda to create a Lambda function and test it by manually invoking it using sample event data. For instructions, see [Create a Lambda Function Authored in Java \(p. 288\)](#).

Example 2: Creating .zip Using Gradle With Local JARs

You may choose not to use the Maven Central repository. Instead have all the dependencies in the project folder. In this case your project folder (`project-dir`) will have the following structure:

```
project-dir/jars/          (all jars go here)
project-dir/build.gradle
project-dir/src/main/java/ (your code goes here)
```

So if your Java code has `example` package and `Hello.java` class, the code will be in the following subdirectory:

```
project-dir/src/main/java/example/Hello.java
```

Your `build.gradle` file should be as follows:

```
apply plugin: 'java'

dependencies {
    compile fileTree(dir: 'jars', include: '*.jar')
}

task buildZip(type: Zip) {
    from compileJava
```

```
    from processResources
    into('lib') {
        from configurations.compileClasspath
    }
}

build.dependsOn buildZip
```

Note that the dependencies specify `fileTree` which identifies `project-dir/jars` as the subdirectory that will include all the required jars.

Now you build the package. Run the following gradle command to build and package the project in a .zip file.

```
project-dir> gradle build
```

Authoring Lambda Functions Using Eclipse IDE and AWS SDK Plugin (Java)

AWS SDK Eclipse Toolkit provides an Eclipse plugin for you to both create a deployment package and also upload it to create a Lambda function. If you can use Eclipse IDE as your development environment, this plugin enables you to author Java code, create and upload a deployment package, and create your Lambda function. For more information, see the [AWS Toolkit for Eclipse Getting Started Guide](#). For an example of using the toolkit for authoring Lambda functions, see [Using AWS Lambda with the AWS Toolkit for Eclipse](#).

AWS Lambda Function Handler in Java

At the time you create a Lambda function you specify a handler that AWS Lambda can invoke when the service executes the Lambda function on your behalf.

Lambda supports two approaches for creating a handler:

- Loading the handler method directly without having to implement an interface. This section describes this approach.
- Implementing standard interfaces provided as part of `aws-lambda-java-core` library (interface approach). For more information, see [Leveraging Predefined Interfaces for Creating Handler \(Java\) \(p. 277\)](#).

The general syntax for the handler is as follows:

```
outputType handler-name(inputType input, Context context) {
    ...
}
```

In order for AWS Lambda to successfully invoke a handler it must be invoked with input data that can be serialized into the data type of the `input` parameter.

In the syntax, note the following:

- `inputType` – The first handler parameter is the input to the handler, which can be event data (published by an event source) or custom input that you provide such as a string or any custom data object. In order for AWS Lambda to successfully invoke this handler, the function must be invoked with input data that can be serialized into the data type of the `input` parameter.

- ***outputType*** – If you plan to invoke the Lambda function synchronously (using the RequestResponse invocation type), you can return the output of your function using any of the supported data types. For example, if you use a Lambda function as a mobile application backend, you are invoking it synchronously. Your output data type will be serialized into JSON.

If you plan to invoke the Lambda function asynchronously (using the Event invocation type), the *outputType* should be void. For example, if you use AWS Lambda with event sources such as Amazon S3 or Amazon SNS, these event sources invoke the Lambda function using the Event invocation type.

- The ***inputType*** and ***outputType*** can be one of the following:
 - Primitive Java types (such as String or int).
 - Predefined AWS event types defined in the aws-lambda-java-events library.

For example S3Event is one of the POJOs predefined in the library that provides methods for you to easily read information from the incoming Amazon S3 event.

- You can also write your own POJO class. AWS Lambda will automatically serialize and deserialize input and output JSON based on the POJO type.

For more information, see [Handler Input/Output Types \(Java\) \(p. 273\)](#).

- You can omit the Context object from the handler method signature if it isn't needed. For more information, see [AWS Lambda Context Object in Java \(p. 281\)](#).

For example, consider the following Java example code.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class Hello implements RequestHandler<Integer, String>{
    public String myHandler(int myCount, Context context) {
        return String.valueOf(myCount);
    }
}
```

In this example input is of type Integer and output is of type String. If you package this code and dependencies, and create your Lambda function, you specify example.Hello::myHandler ([package.class::method-reference](#)) as the handler.

In the example Java code, the first handler parameter is the input to the handler (myHandler), which can be event data (published by an event source such as Amazon S3) or custom input you provide such as an Integer object (as in this example) or any custom data object.

For instructions to create a Lambda function using this Java code, see [Create a Lambda Function Authored in Java \(p. 288\)](#).

Handler Overload Resolution

If your Java code contains multiple methods with same name as the handler name, then AWS Lambda uses the following rules to pick a method to invoke:

1. Select the method with the largest number of parameters.
2. If two or more methods have the same number of parameters, AWS Lambda selects the method that has the Context as the last parameter.

If none or all of these methods have the Context parameter, then the behavior is undefined.

Additional Information

The following topics provide more information about the handler.

- For more information about the handler input and output types, see [Handler Input/Output Types \(Java\) \(p. 273\)](#).
- For information about using predefined interfaces to create a handler, see [Leveraging Predefined Interfaces for Creating Handler \(Java\) \(p. 277\)](#).

If you implement these interfaces, you can validate your handler method signature at compile time.

- If your Lambda function throws an exception, AWS Lambda records metrics in CloudWatch indicating that an error occurred. For more information, see [AWS Lambda Function Errors in Java \(p. 284\)](#).

Handler Input/Output Types (Java)

When AWS Lambda executes the Lambda function, it invokes the handler. The first parameter is the input to the handler which can be event data (published by an event source) or custom input you provide such as a string or any custom data object.

AWS Lambda supports the following input/output types for a handler:

- Simple Java types (AWS Lambda supports the String, Integer, Boolean, Map, and List types)
- POJO (Plain Old Java Object) type
- Stream type (If you do not want to use POJOs or if Lambda's serialization approach does not meet your needs, you can use the byte stream implementation. For more information, see [Example: Using Stream for Handler Input/Output \(Java\) \(p. 276\)](#).)

Handler Input/Output: String Type

The following Java class shows a handler called `myHandler` that uses String type for input and output.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;

public class Hello {
    public String myHandler(String name, Context context) {
        return String.format("Hello %s.", name);
    }
}
```

You can have similar handler functions for other simple Java types.

Note

When you invoke a Lambda function asynchronously, any return value by your Lambda function will be ignored. Therefore you might want to set the return type to void to make this clear in your code. For more information, see [Invoke \(p. 400\)](#).

To test an end-to-end example, see [Create a Lambda Function Authored in Java \(p. 288\)](#).

Handler Input/Output: POJO Type

The following Java class shows a handler called `myHandler` that uses POJOs for input and output.

```
package example;
```

```

import com.amazonaws.services.lambda.runtime.Context;

public class HelloPojo {

    // Define two classes/POJOs for use with Lambda function.
    public static class RequestClass {
        ...
    }

    public static class ResponseClass {
        ...
    }

    public static ResponseClass myHandler(RequestClass request, Context context) {
        String greetingString = String.format("Hello %s, %s.", request.getFirstName(),
request.getLastName());
        return new ResponseClass(greetingString);
    }
}

```

AWS Lambda serializes based on standard bean naming conventions (see [The Java EE 6 Tutorial](#)). You should use mutable POJOs with public getters and setters.

Note

You shouldn't rely on any other features of serialization frameworks such as annotations. If you need to customize the serialization behavior, you can use the raw byte stream to use your own serialization.

If you use POJOs for input and output, you need to provide implementation of the RequestClass and ResponseClass types. For an example, see [Example: Using POJOs for Handler Input/Output \(Java\) \(p. 274\)](#).

Example: Using POJOs for Handler Input/Output (Java)

Suppose your application events generate data that includes first name and last name as shown:

```
{ "firstName": "John", "lastName": "Doe" }
```

For this example, the handler receives this JSON and returns the string "Hello John Doe".

```

public static ResponseClass handleRequest(RequestClass request, Context context){
    String greetingString = String.format("Hello %s, %s.", request.firstName,
request.lastName);
    return new ResponseClass(greetingString);
}

```

To create a Lambda function with this handler, you must provide implementation of the input and output types as shown in the following Java example. The HelloPojo class defines the handler method.

```

package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class HelloPojo implements RequestHandler<RequestClass, ResponseClass>{

    public ResponseClass handleRequest(RequestClass request, Context context){
        String greetingString = String.format("Hello %s, %s.", request.firstName,
request.lastName);

```

```
        return new ResponseClass(greetingString);
    }
}
```

In order to implement the input type, add the following code to a separate file and name it *RequestClass.java*. Place it next to the *HelloPojo.java* class in your directory structure:

```
package example;

public class RequestClass {
    String firstName;
    String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public RequestClass(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public RequestClass() {
    }
}
```

In order to implement the output type, add the following code to a separate file and name it *ResponseClass.java*. Place it next to the *HelloPojo.java* class in your directory structure:

```
package example;

public class ResponseClass {
    String greetings;

    public String getGreetings() {
        return greetings;
    }

    public void setGreetings(String greetings) {
        this.greetings = greetings;
    }

    public ResponseClass(String greetings) {
        this.greetings = greetings;
    }

    public ResponseClass() {
    }
}
```

Note

The get and set methods are required in order for the POJOs to work with AWS Lambda's built in JSON serializer. The constructors that take no arguments are usually not required, however in this example we provided other constructors and therefore we need to explicitly provide the zero argument constructors.

You can upload this code as your Lambda function and test as follows:

- Using the preceding code files, create a deployment package.
- Upload the deployment package to AWS Lambda and create your Lambda function. You can do this using the console or AWS CLI.
- Invoke the Lambda function manually using the console or the CLI. You can use provide sample JSON event data when you manually invoke your Lambda function. For example:

```
{ "firstName": "John", "lastName": "Doe" }
```

For more information, see [Create a Lambda Function Authored in Java \(p. 288\)](#). Note the following differences:

- When you create a deployment package, don't forget the aws-lambda-java-core library dependency.
- When you create the Lambda function, specify `example.HelloPojo::handleRequest` (`package.class::method`) as the handler value.

Example: Using Stream for Handler Input/Output (Java)

If you do not want to use POJOs or if Lambda's serialization approach does not meet your needs, you can use the byte stream implementation. In this case, you can use the `InputStream` and `OutputStream` as the input and output types for the handler. An example handler function is shown:

```
public void handler(InputStream inputStream, OutputStream outputStream, Context context)
    throws IOException{
    ...
}
```

Note that in this case the handler function uses parameters for both the request and response streams.

The following is a Lambda function example that implements the handler that uses `InputStream` and `OutputStream` types for the input and output parameters.

Note

The input payload must be valid JSON but the output stream does not carry such a restriction. Any bytes are supported.

```
package example;

import java.io.InputStream;
import java.io.OutputStream;
import com.amazonaws.services.lambda.runtime.RequestStreamHandler;
import com.amazonaws.services.lambda.runtime.Context;

public class Hello implements RequestStreamHandler{
    public void handler(InputStream inputStream, OutputStream outputStream, Context
        context) throws IOException {
        int letter;
        while((letter = inputStream.read()) != -1)
        {
```

```
        outputStream.write(Character.toUpperCase(letter));
    }
}
```

You can do the following to test the code:

- Using the preceding code, create a deployment package.
- Upload the deployment package to AWS Lambda and create your Lambda function. You can do this using the console or AWS CLI.
- You can manually invoke the code by providing sample input. For example:

```
test
```

Follow instructions provided in the Getting Started. For more information, see [Create a Lambda Function Authored in Java \(p. 288\)](#). Note the following differences:

- When you create a deployment package, don't forget the `aws-lambda-java-core` library dependency.
- When you create the Lambda function, specify `example.Hello::handler` (`package.class::method`) as the handler value.

Leveraging Predefined Interfaces for Creating Handler (Java)

You can use one of the predefined interfaces provided by the AWS Lambda Java core library (`aws-lambda-java-core`) to create your Lambda function handler, as an alternative to writing your own handler method with an arbitrary name and parameters. For more information about handlers, see (see [AWS Lambda Function Handler in Java \(p. 271\)](#)).

You can implement one of the predefined interfaces, `RequestStreamHandler` or `RequestHandler` and provide implementation for the `handleRequest` method that the interfaces provide. You implement one of these interfaces depending on whether you want to use standard Java types or custom POJO types for your handler input/output (where AWS Lambda automatically serializes and deserializes the input and output to Match your data type), or customize the serialization using the `Stream` type.

Note

These interfaces are available in the `aws-lambda-java-core` library.

When you implement standard interfaces, they help you validate your method signature at compile time.

If you implement one of the interfaces, you specify `package.class` in your Java code as the handler when you create the Lambda function. For example, the following is the modified `create-function` CLI command from the getting started. Note that the `--handler` parameter specifies "example.Hello" value:

```
aws lambda create-function \
--region region \
--function-name getting-started-lambda-function-in-java \
--zip-file fileb://deployment-package (zip or jar)
      path \
--role arn:aws:iam::account-id:role/lambda_basic_execution \
--handler example.Hello \
--runtime java8 \
```

```
--timeout 15 \
--memory-size 512
```

The following sections provide examples of implementing these interfaces.

Example 1: Creating Handler with Custom POJO Input/Output (Leverage the RequestHandler Interface)

The example `Hello` class in this section implements the `RequestHandler` interface. The interface defines `handleRequest()` method that takes in event data as input parameter of the `Request` type and returns an POJO object of the `Response` type:

```
public Response handleRequest(Request request, Context context) {
    ...
}
```

The `Hello` class with sample implementation of the `handleRequest()` method is shown. For this example, we assume event data consists of first name and last name.

```
package example;

import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.Context;

public class Hello implements RequestHandler<Request, Response> {

    public Response handleRequest(Request request, Context context) {
        String greetingString = String.format("Hello %s %s.", request.firstName,
        request.lastName);
        return new Response(greetingString);
    }
}
```

For example, if the event data in the `Request` object is:

```
{
    "firstName": "value1",
    "lastName" : "value2"
}
```

The method returns a `Response` object as follows:

```
{
    "greetings": "Hello value1 value2."
}
```

Next, you need to implement the `Request` and `Response` classes. You can use the following implementation for testing:

The `Request` class:

```
package example;

public class Request {
    String firstName;
    String lastName;

    public String getFirstName() {
```

```
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Request(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public Request() {
    }
}
```

The Response class:

```
package example;

public class Response {
    String greetings;

    public String getGreetings() {
        return greetings;
    }

    public void setGreetings(String greetings) {
        this.greetings = greetings;
    }

    public Response(String greetings) {
        this.greetings = greetings;
    }

    public Response() {
    }
}
```

You can create a Lambda function from this code and test the end-to-end experience as follows:

- Using the preceding code, create a deployment package. For more information, see [AWS Lambda Deployment Package in Java \(p. 264\)](#)
- Upload the deployment package to AWS Lambda and create your Lambda function.
- Test the Lambda function using either the console or CLI. You can specify any sample JSON data that conform to the getter and setter in your Request class, for example:

```
{
    "firstName": "John",
    "lastName" : "Doe"
}
```

The Lambda function will return the following JSON in response.

```
{  
    "greetings": "Hello John, Doe."  
}
```

Follow instructions provided in the getting started (see [Create a Lambda Function Authored in Java \(p. 288\)](#)). Note the following differences:

- When you create a deployment package, don't forget the `aws-lambda-java-core` library dependency.
- When you create the Lambda function specify `example.Hello` (`package.class`) as the handler value.

Example 2: Creating Handler with Stream Input/Output (Leverage the RequestStreamHandler Interface)

The `Hello` class in this example implements the `RequestStreamHandler` interface. The interface defines `handleRequest` method as follows:

```
public void handleRequest(InputStream inputStream, OutputStream outputStream, Context  
    context)  
    throws IOException {  
    ...  
}
```

The `Hello` class with sample implementation of the `handleRequest()` handler is shown. The handler processes incoming event data (for example, a string "hello") by simply converting it to uppercase and return it.

```
package example;  
  
import java.io.IOException;  
import java.io.InputStream;  
import java.io.OutputStream;  
  
import com.amazonaws.services.lambda.runtime.RequestStreamHandler;  
import com.amazonaws.services.lambda.runtime.Context;  
  
public class Hello implements RequestStreamHandler {  
    public void handleRequest(InputStream inputStream, OutputStream outputStream, Context  
        context)  
        throws IOException {  
        int letter;  
        while((letter = inputStream.read()) != -1)  
        {  
            outputStream.write(Character.toUpperCase(letter));  
        }  
    }  
}
```

You can create a Lambda function from this code and test the end-to-end experience as follows:

- Use the preceding code to create deployment package.
- Upload the deployment package to AWS Lambda and create your Lambda function.
- Test the Lambda function using either the console or CLI. You can specify any sample string data, for example:

```
"test"
```

The Lambda function will return TEST in response.

Follow instructions provided in the getting started (see [Create a Lambda Function Authored in Java \(p. 288\)](#)). Note the following differences:

- When you create a deployment package, don't forget the `aws-lambda-java-core` library dependency.
- When you create the Lambda function specify `example.Hello` (`package.class`) as the handler value.

AWS Lambda Context Object in Java

When Lambda runs your function, it passes a context object to the [handler \(p. 271\)](#). This object provides methods and properties that provide information about the invocation, function, and execution environment.

Context Methods

- `getRemainingTimeInMillis()` – Returns the number of milliseconds left before the execution times out.
- `getFunctionName()` – Returns the name of the Lambda function.
- `getFunctionVersion()` – Returns the [version \(p. 47\)](#) of the function.
- `getInvokedFunctionArn()` – Returns the Amazon Resource Name (ARN) used to invoke the function. Indicates if the invoker specified a version number or alias.
- `getMemoryLimitInMB()` – Returns the amount of memory configured on the function.
- `getAwsRequestId()` – Returns the identifier of the invocation request.
- `getLogGroupName()` – Returns the log group for the function.
- `getLogStreamName()` – Returns the log stream for the function instance.
- `getIdentity()` – (mobile apps) Returns information about the Amazon Cognito identity that authorized the request.
- `getClientContext()` – (mobile apps) Returns the client context provided to the Lambda invoker by the client application.
- `getLogger()` – Returns the [logger object \(p. 281\)](#) for the function.

AWS Lambda Function Logging in Java

Your Lambda function comes with a CloudWatch Logs log group, with a log stream for each instance of your function. The runtime sends details about each invocation to the log stream, and relays logs and other output from your function's code.

To output logs from your function code, you can use methods on `java.lang.System`, or any logging module that writes to `stdout` or `stderr`. The following example uses `System.out.println`.

Example Hello.java

```
package example;
```

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;

public class Hello {
    public String myHandler(String name, Context context) {
        System.out.println("Event received.");
        return String.format("Hello %s.", name);
    }
}
```

The Lambda console shows log output when you test a function on the function configuration page. To view logs for all invocations, use the CloudWatch Logs console.

To view your Lambda function's logs

1. Open the [Logs page of the CloudWatch console](#).
2. Choose the log group for your function (`/aws/lambda/function-name`).
3. Choose the first stream in the list.

Each log stream corresponds to an [instance of your function \(p. 104\)](#). New streams appear when you update your function and when additional instances are created to handle multiple concurrent invocations. To find logs for specific invocations, you can instrument your function with X-Ray and record details about the request and log stream in the trace. For a sample application that correlates logs and traces with X-Ray, see [Error Processor Sample Application for AWS Lambda \(p. 119\)](#).

To get logs for an invocation from the command line, use the `--log-type` option. The response includes a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

```
$ aws lambda invoke --function-name my-function out --log-type Tail
{
    "StatusCode": 200,
    "LogResult":
    "U1RBULQgUmVxdWVzdElkOjA4N2QwNDRiOC1mMTU0LTEzTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

You can use the `base64` utility to decode the logs.

```
$ aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text | base64 -d
START RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d Version: $LATEST
Processing event...
END RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d
REPORT RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d Duration: 29.40 ms          Billed
Duration: 100 ms      Memory Size: 128 MB      Max Memory Used: 19 MB
```

`base64` is available on Linux, macOS, and [Ubuntu on Windows](#). For macOS, the command is `base64 -D`.

Log groups are not deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.

LambdaLogger

Lambda provides a logger object that you can retrieve from the context object. `LambdaLogger` supports multi-line logs. If you log a string that includes line breaks with `System.out.println`, each line break results in a separate entry in CloudWatch Logs. If you use `LambdaLogger`, you get one entry with multiple lines.

The following example function logs context information.

Example ContextLogger.java

```
package example;
import java.io.InputStream;
import java.io.OutputStream;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;

public class ContextLogger {
    public static void myHandler(InputStream inputStream, OutputStream outputStream,
        Context context) {
        LambdaLogger logger = context.getLogger();
        int letter;
        try {
            while((letter = inputStream.read()) != -1)
            {
                outputStream.write(Character.toUpperCase(letter));
            }
            Thread.sleep(3000); // Intentional delay for testing the
            getRemainingTimeInMillis() result.
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }

        logger.log("Log data from LambdaLogger \n with multiple lines");
        // Print info from the context object
        logger.log("Function name: " + context.getFunctionName());
        logger.log("Max mem allocated: " + context.getMemoryLimitInMB());
        logger.log("Time remaining in milliseconds: " +
        context.getRemainingTimeInMillis());

        // Return the log stream name so you can look up the log later.
        return String.format("Hello %s. log stream = %s", name,
        context.getLogStreamName());
    }
}
```

Dependencies

- `aws-lambda-java-core`

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Java \(p. 264\)](#).

Custom Appender for Log4j 2

AWS Lambda recommends Log4j 2 to provide a custom appender. You can use the custom [Apache log4j](#) appender provided by Lambda for logging from your functions. The custom appender is called `LambdaAppender` and must be used in the `log4j2.xml` file. You must include the `aws-lambda-java-log4j2` artifact (`artifactId:aws-lambda-java-log4j2`) in the deployment package.

Example Hello.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
```

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Hello {
    // Initialize the Log4j logger.
    static final Logger logger = LogManager.getLogger(Hello.class);

    public String myHandler(String name, Context context) {
        logger.error("log data from log4j err.");

        // Return will include the log stream name so you can look
        // up the log later.
        return String.format("Hello %s. log stream = %s", name,
            context.getLogStreamName());
    }
}
```

The example preceding uses the following log4j2.xml file to load properties

Example log4j2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration packages="com.amazonaws.services.lambda.runtime.log4j2">
    <Appenders>
        <Lambda name="Lambda">
            <PatternLayout>
                <pattern>%d{yyyy-MM-dd HH:mm:ss} %X{AWSRequestId} %-5p %c{1}:%L - %m%n</pattern>
            </PatternLayout>
        </Lambda>
    </Appenders>
    <Loggers>
        <Root level="info">
            <AppenderRef ref="Lambda" />
        </Root>
    </Loggers>
</Configuration>
```

Dependencies

- aws-lambda-java-log4j2
- log4j-core
- log4j-api

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [AWS Lambda Deployment Package in Java \(p. 264\)](#).

AWS Lambda Function Errors in Java

If your Lambda function throws an exception, AWS Lambda recognizes the failure and serializes the exception information into JSON and returns it. Following is an example error message:

```
{
    "errorMessage": "Name John Doe is invalid. Exception occurred...",
    "errorType": "java.lang.Exception",
    "stackTrace": [
        "example.Hello.handler(Hello.java:9)",
        "sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)",
```

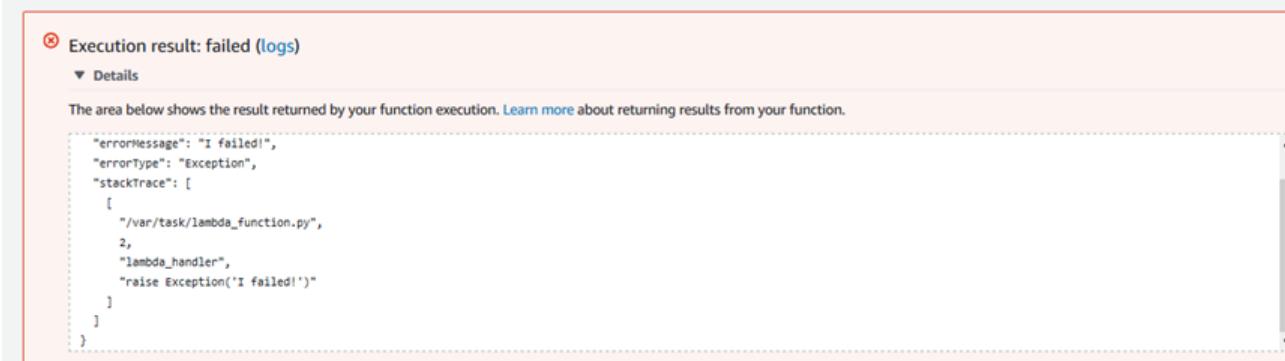
```
"sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)",  
"sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)",  
"java.lang.reflect.Method.invoke(Method.java:497)"  
]  
}
```

Note that the stack trace is returned as the `stackTrace` JSON array of stack trace elements.

The method in which you get the error information back depends on the invocation type that you specified at the time you invoked the function:

- **RequestResponse** invocation type (that is, synchronous execution): In this case, you get the error message back.

For example, if you invoke a Lambda function using the Lambda console, the `RequestResponse` is always the invocation type and the console displays the error information returned by AWS Lambda in the **Execution result** section as shown in the following image.



- **Event** invocation type (that is, asynchronous execution): In this case AWS Lambda does not return anything. Instead, it logs the error information in CloudWatch Logs and CloudWatch metrics.

Depending on the event source, AWS Lambda may retry the failed Lambda function. For example, if Kinesis is the event source for the Lambda function, AWS Lambda retries the failed function until the Lambda function succeeds or the records in the stream expire.

Function Error Handling

You can create custom error handling to raise an exception directly from your Lambda function and handle it directly (Retry or Catch) within an AWS Step Functions State Machine. For more information, see [Handling Error Conditions Using a State Machine](#).

Consider a `CreateAccount` state is a `task` that writes a customer's details to a database using a Lambda function.

- If the task succeeds, an account is created and a welcome email is sent.
- If a user tries to create an account for a username that already exists, the Lambda function raises an error, causing the state machine to suggest a different username and to retry the account-creation process.

The following code samples demonstrate how to do this. Note that custom errors in Java must extend the `Exception` class.

```
package com.example;
```

```

public static class AccountAlreadyExistsException extends Exception {
    public AccountAlreadyExistsException(String message) {
        super(message);
    }
}

package com.example;

import com.amazonaws.services.lambda.runtime.Context;

public class Handler {
    public static void CreateAccount(String name, Context context) throws
    AccountAlreadyExistsException {
        throw new AccountAlreadyExistsException ("Account is in use!");
    }
}

```

You can configure Step Functions to catch the error using a [Catch rule](#). Lambda automatically sets the error name to the fully-qualified class name of the exception at runtime:

```

{
    "StartAt": "CreateAccount",
    "States": {
        "CreateAccount": {
            "Type": "Task",
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:CreateAccount",
            "Next": "SendWelcomeEmail",
            "Catch": [
                {
                    "ErrorEquals": [ "com.example.AccountAlreadyExistsException" ],
                    "Next": "SuggestAccountName"
                }
            ]
        },
        ...
    }
}

```

At runtime, AWS Step Functions catches the error, [transitioning](#) to the `SuggestAccountName` state as specified in the `Next` transition.

Custom error handling makes it easier to create [serverless](#) applications. This feature integrates with all the languages supported by the Lambda [Programming Model \(p. 31\)](#), allowing you to design your application in the programming languages of your choice, mixing and matching as you go.

To learn more about creating your own serverless applications using AWS Step Functions and AWS Lambda, see [AWS Step Functions](#).

Instrumenting Java Code in AWS Lambda

In Java, you can have Lambda emit subsegments to X-Ray to show you information regarding downstream calls to other AWS services made by your function. To take advantage of this capability, include the [AWS X-Ray SDK for Java](#) in your deployment package. No code changes are needed. As long as you are using an AWS SDK version 1.11.48 or later, there is no need to add any additional code lines for downstream calls from your function to be traced.

The AWS SDK will dynamically import the X-Ray SDK to emit subsegments for downstream calls made by your function. By using the X-Ray SDK for Java, you can instrument your code in order to emit custom subsegments and or add annotations to your X-Ray segments.

The following example uses the X-Ray SDK for Java to instrument a Lambda function to emit a custom subsegment and send custom annotation to X-Ray:

```

package uptime;

import java.io.IOException;
import java.time.Instant;
import java.util.HashMap;
import java.util.Map;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;

import com.amazonaws.regions.Regions;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.model.AttributeValue;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.proxies.apache.http.HttpClientBuilder;

public class Hello {
    private static final Log logger = LogFactory.getLog(Hello.class);

    private static final AmazonDynamoDB dynamoClient;
    private static final HttpClient httpClient;

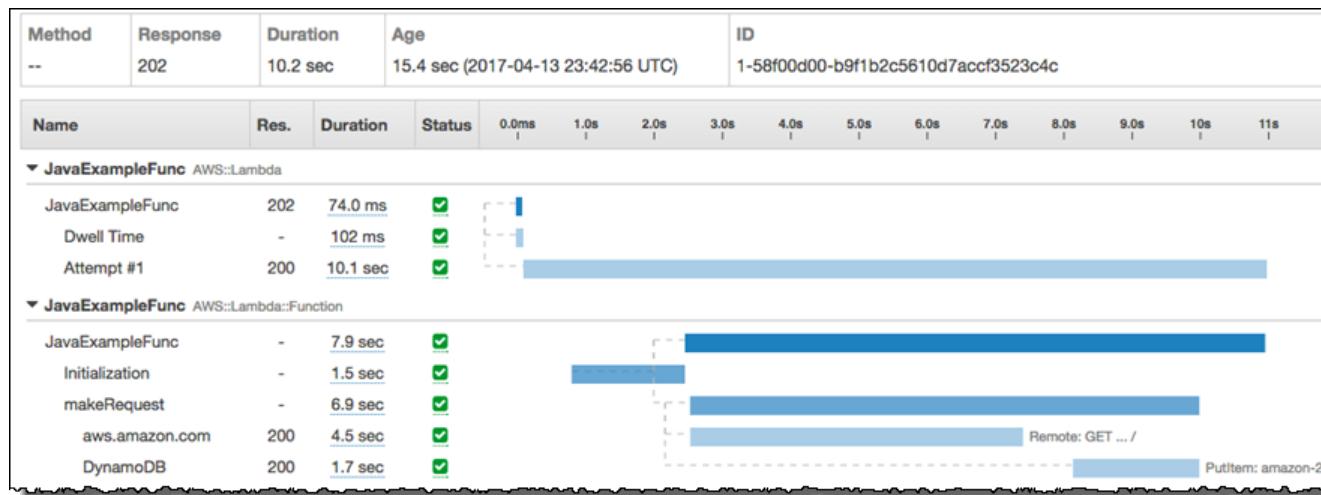
    static {
        dynamoClient =
AmazonDynamoDBClientBuilder.standard().withRegion(Regions.US_EAST_1).build();
        httpClient = HttpClientBuilder.create().build();
    }
    public void checkUptime(Context context) {
        AWSXRay.createSubsegment("makeRequest", (subsegment) -> {

            HttpGet request = new HttpGet("https://aws.amazon.com/");
            boolean is2xx = false;

            try {
                HttpResponse response = httpClient.execute(request);
                is2xx = (response.getStatusLine().getStatusCode() / 100) == 2;
                subsegment.putAnnotation("statusCode",
response.getStatusLine().getStatusCode());
            } catch (IOException ioe) {
                logger.error(ioe);
            }
            Map<String, AttributeValue> item = new HashMap<>();
            item.put("Timestamp", new AttributeValue().withN("" +
Instant.now().getEpochSecond()));
            item.put("2xx", new AttributeValue().withBOOL(is2xx));
            dynamoClient.putItem("amazon-2xx", item);
        });
    }
}

```

Following is what a trace emitted by the code preceding looks like (synchronous invocation):



Create a Lambda Function Authored in Java

The blueprints provide sample code authored either in Python or Node.js. You can easily modify the example using the inline editor in the console. However, if you want to author code for your Lambda function in Java, there are no blueprints provided. Also, there is no inline editor for you to write Java code in the AWS Lambda console.

That means, you must write your Java code and also create your deployment package outside the console. After you create the deployment package, you can use the console to upload the package to AWS Lambda to create your Lambda function. You can also use the console to test the function by manually invoking it.

In this section you create a Lambda function using the following Java code example.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;

public class Hello {
    public String myHandler(int myCount, Context context) {
        LambdaLogger logger = context.getLogger();
        logger.log("received : " + myCount);
        return String.valueOf(myCount);
    }
}
```

The programming model explains how to write your Java code in detail, for example the input/output types AWS Lambda supports. For more information about the programming model, see [Building Lambda Functions with Java \(p. 263\)](#). For now, note the following about this code:

- When you package and upload this code to create your Lambda function, you specify the `example.Hello::myHandler` method reference as the handler.
- The handler in this example uses the `int` type for input and the `String` type for output.

AWS Lambda supports input/output of JSON-serializable types and `InputStream/OutputStream` types. When you invoke this function you will pass a sample `int` (for example, 123).

- You can use the Lambda console to manually invoke this Lambda function. The console always uses the RequestResponse invocation type (synchronous) and therefore you will see the response in the console.
- The handler includes the optional `Context` parameter. In the code we use the `LambdaLogger` provided by the `Context` object to write log entries to CloudWatch logs. For information about using the `Context` object, see [AWS Lambda Context Object in Java \(p. 281\)](#).

First, you need to package this code and any dependencies into a deployment package. Then, you can use the Getting Started exercise to upload the package to create your Lambda function and test using the console. For more information creating a deployment package, see [AWS Lambda Deployment Package in Java \(p. 264\)](#).

Building Lambda Functions with Go

The following sections explain how [common programming patterns and core concepts](#) apply when authoring Lambda function code in [Go](#).

Go Runtimes

Name	Identifier	Operating System
Go 1.x	go1.x	Amazon Linux

Topics

- [AWS Lambda Deployment Package in Go \(p. 290\)](#)
- [AWS Lambda Function Handler in Go \(p. 291\)](#)
- [AWS Lambda Context Object in Go \(p. 295\)](#)
- [AWS Lambda Function Logging in Go \(p. 296\)](#)
- [AWS Lambda Function Errors in Go \(p. 298\)](#)
- [Instrumenting Go Code in AWS Lambda \(p. 300\)](#)
- [Using Environment Variables \(p. 301\)](#)

Additionally, note that AWS Lambda provides the following:

- [github.com/aws/aws-lambda-go/lambda](#): The implementation of the Lambda programming model for Go. This package is used by AWS Lambda to invoke your [AWS Lambda Function Handler in Go \(p. 291\)](#).
- [github.com/aws/aws-lambda-go/lambdacontext](#): Helpers for accessing execution context information from the [AWS Lambda Context Object in Go \(p. 295\)](#).
- [github.com/aws/aws-lambda-go/events](#): This library provides type definitions for common event source integrations.

AWS Lambda Deployment Package in Go

To create a Lambda function you first create a Lambda function deployment package, a .zip file consisting of your code and any dependencies.

After you create a deployment package, you may either upload it directly or upload the .zip file first to an Amazon S3 bucket in the same AWS region where you want to create the Lambda function, and then specify the bucket name and object key name when you create the Lambda function using the console or the AWS CLI.

For Lambda functions written in Go, download the Lambda library for Go by navigating to the Go runtime directory and enter the following command: `go get github.com/aws/aws-lambda-go/lambda`

Then use following command to build, package and deploy a Go Lambda function via the CLI. Note that your `function-name` must match the name of your `Lambda handler` name.

```
GOOS=linux go build lambda_handler.go
zip handler.zip ./lambda_handler
```

```
# --handler is the path to the executable inside the .zip
aws lambda create-function \
--region region \
--function-name lambda-handler \
--memory 128 \
--role arn:aws:iam::account-id:role/execution_role \
--runtime go1.x \
--zip-file fileb://path-to-your-zip-file/handler.zip \
--handler lambda-handler
```

Note

If you are using a non-Linux environment, such as Windows or macOS, ensure that your handler function is compatible with the Lambda [Execution Context](#) by setting the `GOOS` (Go Operating System) environment variable to 'linux' when compiling your handler function code.

Creating a Deployment Package on Windows

To create a .zip that will work on AWS Lambda using Windows, we recommend installing the **build-lambda-zip** tool.

Note

If you have not already done so, you will need to install [git](#) and then add the `git` executable to your Windows %PATH% environment variable.

To download the tool, run the following command:

```
go.exe get -u github.com/aws/aws-lambda-go/cmd/build-lambda-zip
```

Use the tool from your GOPATH. If you have a default installation of Go, the tool will typically be in %USERPROFILE%\Go\bin. Otherwise, navigate to where you installed the Go runtime and do the following:

In cmd.exe, run the following:

```
set GOOS=linux
go build -o main main.go
%USERPROFILE%\Go\bin\build-lambda-zip.exe -o main.zip main
```

In Powershell, run the following:

```
$env:GOOS = "linux"
go build -o main main.go
~\Go\Bin\build-lambda-zip.exe -o main.zip main
```

AWS Lambda Function Handler in Go

A Lambda function written in [Go](#) is authored as a Go executable. In your Lambda function code, you need to include the [github.com/aws/aws-lambda-go/lambda](#) package, which implements the Lambda programming model for Go. In addition, you need to implement handler function code and a `main()` function.

```
package main

import (
    "fmt"
```

```

        "context"
        "github.com/aws/aws-lambda-go/lambda"
    )

type MyEvent struct {
    Name string `json:"name"`
}

func HandleRequest(ctx context.Context, name MyEvent) (string, error) {
    return fmt.Sprintf("Hello %s!", name.Name), nil
}

func main() {
    lambda.Start(HandleRequest)
}

```

Note the following:

- **package main:** In Go, the package containing `func main()` must always be named `main`.
- **import:** Use this to include the libraries your Lambda function requires. In this instance, it includes:
 - **context:** [AWS Lambda Context Object in Go \(p. 295\)](#).
 - **fmt:** The Go [Formatting](#) object used to format the return value of your function.
 - **github.com/aws/aws-lambda-go/lambda:** As mentioned previously, implements the Lambda programming model for Go.
- **func HandleRequest(ctx context.Context, name MyEvent) (string, error):** This is your Lambda handler signature and includes the code which will be executed. In addition, the parameters included denote the following:
 - **ctx context.Context:** Provides runtime information for your Lambda function invocation. `ctx` is the variable you declare to leverage the information available via [AWS Lambda Context Object in Go \(p. 295\)](#).
 - **name MyEvent:** An input type with a variable name of `name` whose value will be returned in the `return` statement.
 - **string, error:** Returns standard `error` information. For more information on custom error handling, see [AWS Lambda Function Errors in Go \(p. 298\)](#).
 - **return fmt.Sprintf("Hello %s!", name), nil:** Simply returns a formatted "Hello" greeting with the name you supplied in the handler signature. `nil` indicates there were no errors and the function executed successfully.
- **func main():** The entry point that executes your Lambda function code. This is required.

By adding `lambda.Start(HandleRequest)` between `func main() {}` code brackets, your Lambda function will be executed.

Note

Per Go language standards, the opening bracket, `{` must be placed directly at end the of the `main` function signature.

Lambda Function Handler Using Structured Types

In the example above, the input type was a simple string. But you can also pass in structured events to your function handler:

```

package main

import (
    "fmt"
    "github.com/aws/aws-lambda-go/lambda"
)

```

```
)  
  
type MyEvent struct {  
    Name string `json:"What is your name?"`  
    Age int     `json:"How old are you?"`  
}  
  
type MyResponse struct {  
    Message string `json:"Answer:"`  
}  
  
func HandleLambdaEvent(event MyEvent) (MyResponse, error) {  
    return MyResponse{Message: fmt.Sprintf("%s is %d years old!", event.Name,  
    event.Age)}, nil  
}  
  
func main() {  
    lambda.Start(HandleLambdaEvent)  
}
```

Your request would then look like this:

```
# request  
{  
    "What is your name?": "Jim",  
    "How old are you?": 33  
}
```

And the response would look like this:

```
# response  
{  
    "Answer": "Jim is 33 years old!"  
}
```

For more information on handling events from AWS event sources, see [aws-lambda-go/events](#).

Valid Handler Signatures

You have several options when building a Lambda function handler in Go, but you must adhere to the following rules:

- The handler must be a function.
- The handler may take between 0 and 2 arguments. If there are two arguments, the first argument must implement `context.Context`.
- The handler may return between 0 and 2 arguments. If there is a single return value, it must implement `error`. If there are two return values, the second value must implement `error`. For more information on implementing error-handling information, see [AWS Lambda Function Errors in Go \(p. 298\)](#).

The following lists valid handler signatures. `TIn` and `TOut` represent types compatible with the `encoding/json` standard library. For more information, see [func Unmarshal](#) to learn how these types are deserialized.

- `func ()`

- `func () error`
- `func (TIn), error`
- `func () (TOut, error)`
- `func (context.Context) error`
- `func (context.Context, TIn) error`
- `func (context.Context) (TOut, error)`
- `func (context.Context, TIn) (TOut, error)`

Using Global State

You can declare and modify global variables that are independent of your Lambda function's handler code. In addition, your handler may declare an `init` function that is executed when your handler is loaded. This behaves the same in AWS Lambda as it does in standard Go programs. A single instance of your Lambda function will never handle multiple events simultaneously. This means, for example, that you may safely change global state, assured that those changes will require a new Execution Context and will not introduce locking or unstable behavior from function invocations directed at the previous Execution Context. For more information, see the following:

- [AWS Lambda Execution Context \(p. 104\)](#)
- [Best Practices for Working with AWS Lambda Functions \(p. 127\)](#)

```
package main

import (
    "log"
    "github.com/aws/aws-lambda-go/lambda"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
    "github.com/aws/aws-sdk-go/aws"
)

var invokeCount = 0
var myObjects []*s3.Object
func init() {
    svc := s3.New(session.New())
    input := &s3.ListObjectsV2Input{
        Bucket: aws.String("examplebucket"),
    }
    result, _ := svc.ListObjectsV2(input)
    myObjects = result.Contents
}

func LambdaHandler() (int, error) {
    invokeCount = invokeCount + 1
    log.Print(myObjects)
    return invokeCount, nil
}

func main() {
```

```
    lambda.Start(LambdaHandler)
}
```

AWS Lambda Context Object in Go

When Lambda runs your function, it passes a context object to the [handler \(p. 291\)](#). This object provides methods and properties with information about the invocation, function, and execution environment.

The Lambda context library provides the following global variables, methods, and properties.

Global Variables

- `FunctionName` – The name of the Lambda function.
- `FunctionVersion` – The [version \(p. 47\)](#) of the function.
- `MemoryLimitInMB` – The amount of memory configured on the function.
- `LogGroupName` – The log group for the function.
- `LogStreamName` – The log stream for the function instance.

Context Methods

- `Deadline` – Returns the date that the execution times out, in Unix time milliseconds.

Context Properties

- `InvokedFunctionArn` – The Amazon Resource Name (ARN) used to invoke the function. Indicates if the invoker specified a version number or alias.
- `AwsRequestId` – The identifier of the invocation request.
- `Identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
- `ClientContext` – (mobile apps) Client context provided to the Lambda invoker by the client application.

Accessing Invoke Context Information

Lambda functions have access to metadata about their environment and the invocation request. This can be accessed at [Package context](#). Should your handler include `context.Context` as a parameter, Lambda will insert information about your function into the context's `Value` property. Note that you need to import the `lambdacontext` library to access the contents of the `context.Context` object.

```
package main

import (
    "context"
    "log"
    "github.com/aws/aws-lambda-go/lambda"
    "github.com/aws/aws-lambda-go/lambdacontext"
)

func CognitoHandler(ctx context.Context) {
    lc, _ := lambdacontext.FromContext(ctx)
    log.Print(lc.Identity.CognitoPoolID)
}
```

```
func main() {
    lambda.Start(CognitoHandler)
}
```

In the example above, `lc` is the variable used to consume the information that the context object captured and `log.Print(lc.Identity.CognitoPoolID)` prints that information, in this case, the `CognitoPoolID`.

Monitoring Execution Time of a Function

The following example introduces how to use the context object to monitor how long it takes to execute your Lambda function. This allows you to analyze performance expectations and adjust your function code accordingly, if needed.

```
package main

import (
    "context"
    "log"
    "time"
    "github.com/aws/aws-lambda-go/lambda"
)

func LongRunningHandler(ctx context.Context) (string, error) {

    deadline, _ := ctx.Deadline()
    deadline = deadline.Add(-100 * time.Millisecond)
    timeoutChannel := time.After(time.Until(deadline))

    for {

        select {

        case <- timeoutChannel:
            return "Finished before timing out.", nil

        default:
            log.Println("hello!")
            time.Sleep(50 * time.Millisecond)
        }
    }
}

func main() {
    lambda.Start(LongRunningHandler)
}
```

AWS Lambda Function Logging in Go

Your Lambda function comes with a CloudWatch Logs log group, with a log stream for each instance of your function. The runtime sends details about each invocation to the log stream, and relays logs and other output from your function's code.

To output logs from your function code, you can use methods on [the `fmt` package](#), or any logging library that writes to `stdout` or `stderr`. The following example uses `fmt.Println`.

```
package main
```

```
import (
    "fmt"
    "github.com/aws/aws-lambda-go/lambda"
)

func HandleRequest() {
    fmt.Println("Hello from Lambda")
}

func main() {
    lambda.Start(HandleRequest)
}
```

For more detailed logs, use [the log package](#).

```
package main

import (
    "log"
    "github.com/aws/aws-lambda-go/lambda"
)

func HandleRequest() {
    log.Println("Event received.")
}

func main() {
    lambda.Start(HandleRequest)
}
```

The output from `log` includes the timestamp and request ID.

The Lambda console shows log output when you test a function on the function configuration page. To view logs for all invocations, use the CloudWatch Logs console.

To view your Lambda function's logs

1. Open the [Logs page of the CloudWatch console](#).
2. Choose the log group for your function (`/aws/lambda/function-name`).
3. Choose the first stream in the list.

Each log stream corresponds to an [instance of your function \(p. 104\)](#). New streams appear when you update your function and when additional instances are created to handle multiple concurrent invocations. To find logs for specific invocations, you can instrument your function with X-Ray and record details about the request and log stream in the trace. For a sample application that correlates logs and traces with X-Ray, see [Error Processor Sample Application for AWS Lambda \(p. 119\)](#).

To get logs for an invocation from the command line, use the `--log-type` option. The response includes a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

```
$ aws lambda invoke --function-name my-function out --log-type Tail
{
    "StatusCode": 200,
    "LogResult": "U1RBULQgUmVxdWVzdElkOiaA4N2QwNDRiOC1mMTU0LTEzZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

You can use the `base64` utility to decode the logs.

```
$ aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text | base64 -d
START RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d Version: $LATEST
Processing event...
END RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d
REPORT RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d Duration: 29.40 ms      Billed
Duration: 100 ms          Memory Size: 128 MB     Max Memory Used: 19 MB
```

`base64` is available on Linux, macOS, and [Ubuntu on Windows](#). For macOS, the command is `base64 -D`.

Log groups are not deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.

AWS Lambda Function Errors in Go

You can create custom error handling to raise an exception directly from your Lambda function and handle it directly.

The following code samples demonstrate how to do this. Note that custom errors in Go must import the `errors` module.

```
package main

import (
    "errors"
    "github.com/aws/aws-lambda-go/lambda"
)

func OnlyErrors() error {
    return errors.New("something went wrong!")
}

func main() {
    lambda.Start(OnlyErrors)
}
```

Which returns the following:

```
{
    "errorMessage": "something went wrong!",
    "errorType": "errorString"
}
```

Function Error Handling

You can create custom error handling to raise an exception directly from your Lambda function and handle it directly (Retry or Catch) within an AWS Step Functions State Machine. For more information, see [Handling Error Conditions Using a State Machine](#).

Consider a `CreateAccount` state is a `task` that writes a customer's details to a database using a Lambda function.

- If the task succeeds, an account is created and a welcome email is sent.
- If a user tries to create an account for a username that already exists, the Lambda function raises an error, causing the state machine to suggest a different username and to retry the account-creation process.

The following code samples demonstrate how to do this.

```
package main

type CustomError struct {}

func (e *CustomError) Error() string {
    return "bad stuff happened..."
}

func MyHandler() (string, error) {
    return "", &CustomError{}
}
```

At runtime, AWS Step Functions catches the error, [transitioning](#) to the `SuggestAccountName` state as specified in the `Next` transition.

Custom error handling makes it easier to create [serverless](#) applications. This feature integrates with all the languages supported by the Lambda [Programming Model](#) (p. 31), allowing you to design your application in the programming languages of your choice, mixing and matching as you go.

To learn more about creating your own serverless applications using AWS Step Functions and AWS Lambda, see [AWS Step Functions](#).

Handling Unexpected Errors

Lambda functions can fail for reasons beyond your control, such as network outages. These are exceptional circumstances. In Go, `panic` addresses these issues. If your code panics, Lambda will attempt to capture the error and serialize it into the standard error json format. Lambda will also attempt to insert the value of the panic into the function's CloudWatch logs. After returning the response, Lambda will re-create the function automatically. If you find it necessary, you can include the `panic` function in your code to customize the error response.

```
package main

import (
    "errors"

    "github.com/aws/aws-lambda-go/lambda"
)

func handler(string) (string, error) {
    panic(errors.New("Something went wrong"))
}

func main() {
    lambda.Start(handler)
}
```

Which would return the following stack in json:

```
{
    "errorMessage": "Something went wrong",
    "errorType": "errorString",
    "stackTrace": [
        {
            "path": "github.com/aws/aws-lambda-go/lambda/function.go",
            "line": 27,
            "label": "(*Function).Invoke.function"
        },
    ]}
```

```
...  
]  
}
```

Instrumenting Go Code in AWS Lambda

You can use the [X-Ray SDK for Go](#) with your Lambda function. If your handler includes [AWS Lambda Context Object in Go \(p. 295\)](#) as its first argument, that object can be passed to the X-Ray SDK. Lambda passes values through this context that the SDK can use to attach subsegments to the Lambda invoke service segment. Subsegments created with the SDK will appear as a part of your Lambda traces.

Installing the X-Ray SDK for Go

Use the following command to install the X-Ray SDK for Go. (The SDK's non-testing dependencies will be included).

```
go get -u github.com/aws/aws-xray-sdk-go/...
```

If you want to include the test dependencies, use the following command:

```
go get -u -t github.com/aws/aws-xray-sdk-go/...
```

You can also use [Glide](#) to manage dependencies.

```
glide install
```

Configuring the X-Ray SDK for Go

The following code sample illustrates how to configure the X-Ray SDK for Go in your Lambda function:

```
import (  
    "github.com/aws/aws-xray-sdk-go/xray"  
)  
func myHandlerFunction(ctx context.Context, sample string) {  
    xray.Configure(xray.Config{  
        LogLevel:      "info",           // default  
        ServiceVersion: "1.2.3",  
    })  
    ... //remaining handler code  
}
```

Create a subsegment

The following code illustrates how to start a subsegment:

```
// Start a subsegment  
ctx, subSeg := xray.BeginSubsegment(ctx, "subsegment-name")  
// ...  
// Add metadata or annotation here if necessary  
// ...  
subSeg.Close(nil)
```

Capture

The following code illustrates how to trace and capture a critical code path:

```
func criticalSection(ctx context.Context) {
    // This example traces a critical code path using a custom subsegment
    xray.Capture(ctx, "MyService.criticalSection", func(ctx1 context.Context) error {
        var err error

        section.Lock()
        result := someLockedResource.Go()
        section.Unlock()

        xray.AddMetadata(ctx1, "ResourceResult", result)
    })
}
```

Tracing HTTP Requests

You can also use the `xray.Client()` method if you want to trace an HTTP client, as shown below:

```
func myFunction (ctx context.Context) ([]byte, error) {
    resp, err := ctxhttp.Get(ctx, xray.Client(nil), "https://aws.amazon.com")
    if err != nil {
        return nil, err
    }
    return ioutil.ReadAll(resp.Body), nil
}
```

Using Environment Variables

To access [AWS Lambda Environment Variables \(p. 40\)](#) in Go, use the `Getenv` function.

The following explains how to do this. Note that the function imports the `fmt` package to format the printed results and the `os` package, a platform-independent system interface that allows you to access environment variables.

```
package main

import (
    "fmt"
    "os"
    "github.com/aws/aws-lambda-go/lambda"
)

func main() {
    fmt.Printf("%s is %. years old\n", os.Getenv("NAME"), os.Getenv("AGE"))
}
```

Lambda configures the following environment variables by default: [Environment Variables Available to Lambda Functions \(p. 103\)](#).

Building Lambda Functions with C#

The following sections explain how [common programming patterns and core concepts](#) apply when authoring Lambda function code in C#.

.NET Runtimes

Name	Identifier	Languages	Operating System
.NET Core 2.1	dotnetcore2.1	C# PowerShell Core 6.0	Amazon Linux
.NET Core 1.0	dotnetcore1.0	C#	Amazon Linux

Topics

- [AWS Lambda Deployment Package in C# \(p. 302\)](#)
- [AWS Lambda Function Handler in C# \(p. 311\)](#)
- [AWS Lambda Context Object in C# \(p. 315\)](#)
- [AWS Lambda Function Logging in C# \(p. 315\)](#)
- [AWS Lambda Function Errors in C# \(p. 317\)](#)

Additionally, note that AWS Lambda provides the following:

- **Amazon.Lambda.Core** – This library provides a static Lambda logger, serialization interfaces and a context object. The Context object ([AWS Lambda Context Object in C# \(p. 315\)](#)) provides runtime information about your Lambda function.
- **Amazon.Lambda.Serialization.Json** – This is an implementation of the serialization interface in **Amazon.Lambda.Core**.
- **Amazon.Lambda.Logging.AspNetCore** – This provides a library for logging from ASP.NET.
- Event objects (POCOs) for several AWS services, including:
 - **Amazon.Lambda.APIGatewayEvents**
 - **Amazon.Lambda.CognitoEvents**
 - **Amazon.Lambda.ConfigEvents**
 - **Amazon.Lambda.DynamoDBEvents**
 - **Amazon.Lambda.KinesisEvents**
 - **Amazon.Lambda.S3Events**
 - **Amazon.Lambda.SQSEvents**
 - **Amazon.Lambda.SNSEvents**

These packages are available at [Nuget Packages](#).

AWS Lambda Deployment Package in C#

A .NET Core Lambda deployment package is a zip file of your function's compiled assembly along with all of its assembly dependencies. The package also contains a `proj.deps.json` file. This signals to the .NET Core runtime all of your function's dependencies and a `proj.runtimeconfig.json` file, which is used to configure the .NET Core runtime. The .NET CLI's `publish` command can

create a folder with all of these files, but by default the `proj.runtimeconfig.json` will not be included because a Lambda project is typically configured to be a class library. To force the `proj.runtimeconfig.json` to be written as part of the publish process, pass in the command line argument: `/p:GenerateRuntimeConfigurationFiles=true` to the publish command.

Note

Although it is possible to create the deployment package with the `dotnet publish` command, we suggest you create the deployment package with either the [AWS Toolkit for Visual Studio \(p. 310\)](#) or the [.NET Core CLI \(p. 303\)](#). These are tools optimized specifically for Lambda to ensure the `lambda-project.runtimeconfig.json` file exists and optimizes the package bundle, including the removal of any non-Linux-based dependencies.

.NET Core CLI

The .NET Core CLI offers a cross-platform way for you to create .NET-based Lambda applications. This section assumes you have installed the .NET Core CLI. If you haven't, do so [here](#).

In the .NET CLI, you use the `new` command to create .NET projects from a command line. This is particularly useful if you want to create a platform-independent project outside of Visual Studio. To view a list of the available project types, open a command line and navigate to where you installed the .NET Core runtime and enter the following:

```
dotnet new -all
```

You should see the following:

```
dotnet new -all
Usage: new [options]

Options:
  -h, --help           Displays help for this command.
  -l, --list            Lists templates containing the specified name. If no name is
                        specified, lists all templates.
  -n, --name            The name for the output being created. If no name is specified, the
                        name of the current directory is used.
  -o, --output          Location to place the generated output.
  -i, --install         Installs a source or a template pack.
  -u, --uninstall       Uninstalls a source or a template pack.
  --nuget-source        Specifies a NuGet source to use during install.
  --type                Filters templates based on available types. Predefined values are
                        "project", "item" or "other".
  --force               Forces content to be generated even if it would change existing
                        files.
  -lang, --language     Filters templates based on language and specifies the language of the
                        template to create.
```

Templates	Short Name	Language	Tags
<hr/>			
Console Application	console	[C#], F#, VB	
Common/Console			
Class library	classlib	[C#], F#, VB	
Common/Library			
Unit Test Project	mstest	[C#], F#, VB	
Test/MSTest			
xUnit Test Project	xunit	[C#], F#, VB	
Test/xUnit			
Razor Page	page	[C#]	Web/
ASP.NET			
MVC ViewImports	viewimports	[C#]	Web/
ASP.NET			

MVC ViewStart	viewstart	[C#]	Web/
ASP.NET			
ASP.NET Core Empty	web	[C#], F#	Web/
Empty			
ASP.NET Core Web App (Model-View-Controller)	mvc	[C#], F#	Web/
MVC			
ASP.NET Core Web App	razor	[C#]	Web/
MVC/Razor Pages			
ASP.NET Core with Angular	angular	[C#]	Web/
MVC/SPA			
ASP.NET Core with React.js	react	[C#]	Web/
MVC/SPA			
ASP.NET Core with React.js and Redux	reactredux	[C#]	Web/
MVC/SPA			
Razor Class Library	razorclasslib	[C#]	Web/
Razor/Library/Razor Class Library			
ASP.NET Core Web API	webapi	[C#], F#	Web/
WebAPI			
global.json file	globaljson		
Config			
NuGet Config	nugetconfig		
Config			
Web Config	webconfig		
Config			
Solution File	sln		
Solution			
Examples:			
dotnet new mvc --auth Individual			
dotnet new viewstart			
dotnet new --help			

So, for example, if you wanted to create a console project, you would do the following:

1. Make a directory where your project will be created using the following command: `mkdir example`
2. Navigate to that directory using the following command: `cd example`
3. Enter the following command: `dotnet new console -o myproject`

This will create the following files in your `example` directory:

- `Program.cs`, which is where you write your Lambda function code.
- `MyProject.csproj`, an XML file that lists the files and dependencies that comprise your .NET application.

AWS Lambda offers additional templates via the [Amazon.Lambda.Templates](#) nuget package. To install this package, run the following command:

```
dotnet new -i Amazon.Lambda.Templates
```

Once the install is complete, the Lambda templates show up as part of `dotnet new`. To verify this, again run the following command:

```
dotnet new -all
```

You should now see the following:

```
dotnet new -all
Usage: new [options]
```

```
Options:
```

```

-h, --help           Displays help for this command.
-l, --list            Lists templates containing the specified name. If no name is
specified, lists all templates.
-n, --name            The name for the output being created. If no name is specified, the
name of the current directory is used.
-o, --output           Location to place the generated output.
-i, --install           Installs a source or a template pack.
-u, --uninstall         Uninstalls a source or a template pack.
--nuget-source        Specifies a NuGet source to use during install.
--type                Filters templates based on available types. Predefined values are
"project", "item" or "other".
--force               Forces content to be generated even if it would change existing
files.
-lang, --language      Filters templates based on language and specifies the language of the
template to create.

```

Templates		Short Name
Language	Tags	
Order Flowers Chatbot Tutorial	[C#]	lambda.OrderFlowersChatbot
Lambda Detect Image Labels	[C#], F#	lambda.DetectImageLabels
Lambda Empty Function	[C#], F#	lambda.EmptyFunction
Lex Book Trip Sample	[C#]	lambda.LexBookTripSample
Lambda Simple DynamoDB Function	[C#], F#	lambda.DynamoDB
Lambda Simple Kinesis Firehose Function	[C#]	lambda.KinesisFirehose
Lambda Simple Kinesis Function	[C#], F#	lambda.Kinesis
Lambda Simple S3 Function	[C#], F#	lambda.S3
Lambda ASP.NET Core Web API	[C#], F#	serverless.AspNetCoreWebAPI
Lambda ASP.NET Core Web Application with Razor Pages	[C#]	serverless.AspNetCoreWebApp
Serverless Detect Image Labels	[C#], F#	serverless.DetectImageLabels
Lambda DynamoDB Blog API	[C#]	serverless.DynamoDBBlogAPI
Lambda Empty Serverless	[C#], F#	serverless.EmptyServerless
Lambda Giraffe Web App	F#	serverless.Giraffe
Serverless Simple S3 Function	[C#], F#	serverless.S3
Step Functions Hello World	serverless.StepFunctionsHelloWorld [C#], F#	AWS/Lambda/Serverless
Console Application	[C#], F#, VB	console
Class library	[C#], F#, VB	classlib
Unit Test Project	[C#], F#, VB	mstest
xUnit Test Project	[C#], F#, VB	xunit
Razor Page	[C#]	page
MVC ViewImports	[C#]	viewimports
MVC ViewStart	[C#]	viewstart

ASP.NET Core Empty		web
[C#], F#	Web/Empty	
ASP.NET Core Web App (Model-View-Controller)		mvc
[C#], F#	Web/MVC	
ASP.NET Core Web App		razor
[C#]	Web/MVC/Razor Pages	
ASP.NET Core with Angular		angular
[C#]	Web/MVC/SPA	
ASP.NET Core with React.js		react
[C#]	Web/MVC/SPA	
ASP.NET Core with React.js and Redux		reactredux
[C#]	Web/MVC/SPA	
Razor Class Library		razorclasslib
[C#]	Web/Razor/Library/Razor Class Library	
ASP.NET Core Web API		webapi
[C#], F#	Web/WebAPI	
global.json file		globaljson
	Config	
NuGet Config		nugetconfig
	Config	
Web Config		webconfig
	Config	
Solution File		sln
	Solution	
Examples:		
dotnet new mvc --auth Individual		
dotnet new viewimports --namespace		
dotnet new --help		

To examine details about a particular template, use the following command:

```
dotnet new lambda.EmptyFunction --help
```

Note the following:

-p --profile	The AWS credentials profile set in aws-lambda-tools-defaults.json and used as the default profile when interacting with AWS.
string - Optional	
-r --region	The AWS region set in aws-lambda-tools-defaults.json and used as the default region when interacting with AWS.
string - Optional	

These are optional values you can set when you create your Lambda function and will then be automatically written to the aws-lambda-tools-defaults.json file, which is built as part of the function-creation process. The following explains what they mean:

- **--profile** – Your [execution role \(p. 9\)](#).
- **--region** – The Region in which your function will reside.

For example, to create a Lambda function, run the following command, substituting the values of the **--region** parameter with the region of your choice and **--profile** with your IAM profile:

Note

For more information on Lambda function requirements, see [CreateFunction \(p. 355\)](#)

```
dotnet new lambda.EmptyFunction --name MyFunction --profile default --region region
```

This should create a directory structure similar to the following:

```
<dir>myfunction
  /src/myfunction
  /test/myfunction
```

Under the `src/myfunction` directory, examine the following files:

- **aws-lambda-tools-defaults.json:** This is where you specify the command line options when deploying your Lambda function. For example:

```
"profile": "iam_profile",
  "region" : "region",
  "configuration" : "Release",
  "framework" : "netcoreapp2.1",
  "function-runtime": "dotnetcore2.1",
  "function-memory-size" : 256,
  "function-timeout" : 30,
  "function-handler" : "MyFunction::MyFunction.FunctionHandler"
```

- **Function.cs:** Your Lambda handler function code. It's a C# template that includes the default `Amazon.Lambda.Core` library and a default `LambdaSerializer` attribute. For more information on serialization requirements and options, see [Serializing Lambda Functions \(p. 309\)](#). It also includes a sample function that you can edit to apply your Lambda function code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted into
// a .NET class.
[assembly: LambdaSerializer(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]

namespace MyFunction
{
    public class Function
    {

        public string FunctionHandler1(string input, ILambdaContext context)
        {
            return input?.ToUpper();
        }
    }
}
```

- **MyFunction.csproj:** An [MSBuild](#) file that lists the files and assemblies that comprise your application.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="1.0.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.Json" Version="1.3.0" />
  </ItemGroup>

</Project>
```

- **Readme:** Use this file to document your Lambda function.

Under the `myfunction/test` directory, examine the following files:

- **myFunction.Tests.csproj:** As noted above, this is an [MSBuild](#) file that lists the files and assemblies that comprise your test project. Note also that it includes the `Amazon.Lambda.Core` library, allowing you to seamlessly integrate any Lambda templates required to test your function.

```
<Project Sdk="Microsoft.NET.Sdk">
  ...
  <PackageReference Include="Amazon.Lambda.Core" Version="1.0.0" />
  ...
```

- **FunctionTest.cs:** The same C# code template file that it is included in the `src` directory. Edit this file to mirror your function's production code and test it before uploading your Lambda function to a production environment.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

using Xunit;
using Amazon.Lambda.Core;
using Amazon.Lambda.TestUtilities;

using MyFunction;

namespace MyFunction.Tests
{
    public class FunctionTest
    {
        [Fact]
        public void TestToUpperFunction()
        {

            // Invoke the lambda function and confirm the string was upper cased.
            var function = new Function();
            var context = new TestLambdaContext();
            var upperCase = function.FunctionHandler("hello world", context);

            Assert.Equal("HELLO WORLD", upperCase);
        }
    }
}
```

Once your function has passed its tests, you can build and deploy using the `Amazon.Lambda.Tools .NET Core Global Tool`. To install the .NET Core Global Tool run the following command.

```
dotnet tool install -g Amazon.Lambda.Tools
```

If you already have the tool installed you can make sure you are using the latest version with the following command.

```
dotnet tool update -g Amazon.Lambda.Tools
```

For more information about the `Amazon.Lambda.Tools .NET Core Global` see its [GitHub repository](#).

With the `Amazon.Lambda.Tools` installed you can deploy your function with the following command:

```
dotnet lambda deploy-function MyFunction --function-role role
```

After deployment, you can re-test it in a production environment with the following command and pass in a different value to your Lambda function handler:

```
dotnet lambda invoke-function MyFunction --payload "Just Checking If Everything is OK"
```

Presuming everything was successful, you should see the following:

```
dotnet lambda invoke-function MyFunction --payload "Just Checking If Everything is OK"
Payload:
"JUST CHECKING IF EVERYTHING IS OK"

Log Tail:
START RequestId: id Version: $LATEST
END RequestId: id
REPORT RequestId: id Duration: 0.99 ms          Billed Duration: 100 ms      Memory Size:
256 MB     Max Memory Used: 12 MB
```

Serializing Lambda Functions

For any Lambda functions that use input or output types other than a `Stream` object, you will need to add a serialization library to your application. You can do this in the following ways:

- Use Json.NET. Lambda will provide an implementation for JSON serializer using JSON.NET as a [NuGet](#) package.
- Create your own serialization library by implementing the `ILambdaSerializer` interface, which is available as part of the `Amazon.Lambda.Core` library. The interface defines two methods:
 - `T Deserialize<T>(Stream requestStream);`

You implement this method to deserialize the request payload from the `Invoke` API into the object that is passed to the Lambda function handler.

- `T Serialize<T>(T response, Stream responseStream);`

You implement this method to serialize the result returned from the Lambda function handler into the response payload that is returned by the `Invoke` API.

You use whichever serializer you wish by adding it as a dependency to your `MyProject.csproj` file.

```
...
<ItemGroup>
  <PackageReference Include="Amazon.Lambda.Core" Version="1.0.0" />
  <PackageReference Include="Amazon.Lambda.Serialization.Json" Version="1.3.0" />
</ItemGroup>
```

You then add it to your `AssemblyInfo.cs` file. For example, if you are using the default Json.NET serializer, this is what you would add:

```
[assembly:LambdaSerializer(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]
```

Note

You can define a custom serialization attribute at the method level, which will override the default serializer specified at the assembly level. For more information, see [Handling Standard Data Types \(p. 312\)](#).

AWS Toolkit for Visual Studio

You can build .NET-based Lambda applications using the Lambda plugin to the [AWS Toolkit for Visual Studio](#). The plugin is available as part of a [NuGet](#) package.

Step 1: Create and Build a Project

1. Launch Microsoft Visual Studio and choose **New project**.
 - a. From the **File** menu, choose **New**, and then choose **Project**.
 - b. In the **New Project** window, choose **AWS Lambda Project (.NET Core)** and then choose **OK**.
 - c. In the **Select Blueprint** window, you will be presented with the option of selecting from a list of sample applications that will provide you with sample code to get started with creating a .NET-based Lambda application.
 - d. To create a Lambda application from scratch, choose **Blank Function** and then choose **Finish**.
2. Examine the `aws-lambda-tools-defaults.json` file, which is created as part of your project. You can set the options in this file, which is read by the Lambda tooling by default. The project templates created in Visual Studio set many of these fields with default values. Note the following fields:
 - **profile** – Your [execution role](#) (p. 9)
 - **function-handler** – This is where the `function_handler` is specified, which is why you don't have to set it in the wizard. However, whenever you rename the `Assembly`, `Namespace`, `Class` or `Function` in your function code, you will need to update the corresponding fields in the `aws-lambda-tools-defaults.json` file.

```
{  
    "profile": "iam-execution-profile",  
    "region": "region",  
    "configuration": "Release",  
    "framework": "netcoreapp2.1",  
    "function-runtime": "dotnetcore2.1",  
    "function-memory-size": 256,  
    "function-timeout": 30,  
    "function-handler": "Assembly::Namespace.Class::Function"  
}
```

3. Open the `Function.cs` file. You will be provided with a template to implement your Lambda function handler code.

```
using System;  
using Amazon.Lambda.Core;  
using Amazon.Lambda.Serialization;  
  
// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.  
[assembly: LambdaSerializerAttribute(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]  
  
namespace AWSLambda  
{  
    public class LambdaFunction  
    {  
  
        /// <summary>  
        /// A simple function that takes a string and does a ToUpper  
        /// </summary>  
        /// <param name="input"></param>  
        /// <param name="context"></param>  
        /// <returns></returns>  
        public string FunctionHandler(string input, ILambdaContext context)  
        {  
            return input?.ToUpper();  
        }  
    }  
}
```

4. Once you have written the code that represents your Lambda function, you can upload it by right-clicking the **Project** node in your application and then choosing **Publish to AWS Lambda**.
5. In the **Upload Lambda Function** window, type a name for the function or select a previously published function to republish. Then choose **Next**
6. In the **Advanced Function Details** window, do the following:
 - Specify the **Role Name**: the IAM role mentioned previously.
 - (Optional) In **Environment**: specify any environment variables you wish to use. For more information, see [AWS Lambda Environment Variables \(p. 40\)](#).
 - (Optional) Specify the **Memory (MB)**: or **Timeout (Secs)**: configurations.
 - (Optional) Specify any **VPC**: configurations if your Lambda function needs to access resources running inside a VPC. For more information, see [Configuring a Lambda Function to Access Resources in an Amazon VPC \(p. 68\)](#).
 - Choose **Next** and then choose **Upload** to deploy your application.

For more information, see [Deploying an AWS Lambda Project with the .NET Core CLI](#).

AWS Lambda Function Handler in C#

When you create a Lambda function, you specify a handler that AWS Lambda can invoke when the service executes the function on your behalf.

You define a Lambda function handler as an instance or static method in a class. If you want access to the Lambda context object, it is available by defining a method parameter of type *ILambdaContext*, an interface you can use to access information about the current execution, such as the name of the current function, the memory limit, execution time remaining, and logging.

```
returnType handler-name(inputType input, ILambdaContext context) {  
    ...  
}
```

In the syntax, note the following:

- *inputType* – The first handler parameter is the input to the handler, which can be event data (published by an event source) or custom input that you provide such as a string or any custom data object.
- *returnType* – If you plan to invoke the Lambda function synchronously (using the `RequestResponse` invocation type), you can return the output of your function using any of the supported data types. For example, if you use a Lambda function as a mobile application backend, you are invoking it synchronously. Your output data type will be serialized into JSON.

If you plan to invoke the Lambda function asynchronously (using the `Event` invocation type), the `returnType` should be `void`. For example, if you use AWS Lambda with event sources such as Amazon S3 or Amazon SNS, these event sources invoke the Lambda function using the `Event` invocation type.

Handling Streams

Only the `System.IO.Stream` type is supported as an input parameter by default.

For example, consider the following C# example code.

```
using System.IO;

namespace Example
{
    public class Hello
    {
        public Stream MyHandler(Stream stream)
        {
            //function logic
        }
    }
}
```

In the example C# code, the first handler parameter is the input to the handler (MyHandler), which can be event data (published by an event source such as Amazon S3) or custom input you provide such as a `Stream` (as in this example) or any custom data object. The output is of type `Stream`.

Handling Standard Data Types

All other types, as listed below, require you to specify a serializer.

- Primitive .NET types (such as `string` or `int`).
- Collections and maps - `IList`, `IEnumerable`, `IList<T>`, `Array`, `IDictionary`, `IDictionary<TKey, TValue>`
- POCO types (Plain old CLR objects)
- Predefined AWS event types
- For asynchronous invocations the return-type will be ignored by Lambda. The return type may be set to `void` in such cases.
- If you are using .NET asynchronous programming, the return type can be `Task` and `Task<T>` types and use `async` and `await` keywords. For more information, see [Using Async in C# Functions with AWS Lambda \(p. 314\)](#).

Unless your function input and output parameters are of type `System.IO.Stream`, you will need to serialize them. AWS Lambda provides a default serializer that can be applied at the assembly or method level of your application, or you can define your own by implementing the `ILambdaSerializer` interface provided by the `Amazon.Lambda.Core` library. For more information, see [AWS Lambda Deployment Package in C# \(p. 302\)](#).

To add the default serializer attribute to a method, first add a dependency on `Amazon.Lambda.Serialization.Json` in your `project.json` file.

```
{
    "version": "1.0.0-*",
    "dependencies": {
        "Microsoft.NETCore.App": {
            "type": "platform",
            "version": "1.0.1"
        },
        "Amazon.Lambda.Serialization.Json": "1.3.0"
    },
    "frameworks": {
        "netcoreapp1.0": {
            "imports": "dnxcore50"
        }
    }
}
```

The example below illustrates the flexibility you can leverage by specifying the default Json.NET serializer on one method and another of your choosing on a different method:

```
public class ProductService{
    [LambdaSerializer(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]
    public Product DescribeProduct(DescribeProductRequest request)
    {
        return catalogService.DescribeProduct(request.Id);
    }

    [LambdaSerializer(typeof(MyJsonSerializer))]
    public Customer DescribeCustomer(DescribeCustomerRequest request)
    {
        return customerService.DescribeCustomer(request.Id);
    }
}
```

Handler Signatures

When creating Lambda functions, you have to provide a handler string that tells AWS Lambda where to look for the code to invoke. In C#, the format is:

ASSEMBLY::TYPE::METHOD where:

- **ASSEMBLY** is the name of the .NET assembly file for your application. When using the .NET Core CLI to build your application, if you haven't set the assembly name using the `buildOptions.outputName` setting in `project.json`, the **ASSEMBLY** name will be the name of the folder that contains your `project.json` file. For more information, see [.NET Core CLI \(p. 303\)](#). In this case, let's assume the folder name is `HelloWorldApp`.
- **TYPE** is the full name of the handler type, which consists of the **Namespace** and the **ClassName**. In this case `Example.Hello`.
- **METHOD** is name of the function handler, in this case `MyHandler`.

Ultimately, the signature will be of this format: **Assembly::Namespace.ClassName::MethodName**

Again, consider the following example:

```
using System.IO;

namespace Example
{
    public class Hello
    {
        public Stream MyHandler(Stream stream)
        {
            //function logic
        }
    }
}
```

The handler string would be: `HelloWorldApp::Example.Hello::MyHandler`

Important

If the method specified in your handler string is overloaded, you must provide the exact signature of the method Lambda should invoke. AWS Lambda will reject an otherwise valid signature if the resolution would require selecting among multiple (overloaded) signatures.

Lambda Function Handler Restrictions

Note that there are some restrictions on the handler signature.

- It may not be `unsafe` and use pointer types in the handler signature, though `unsafe` context can be used inside the handler method and its dependencies. For more information, see [unsafe \(C# Reference\)](#).
- It may not pass a variable number of parameters using the `params` keyword, or use `ArgIterator` as an input or return parameter which is used to support variable number of parameters.
- The handler may not be a generic method (e.g. `IList<T> Sort<T>(IList<T> input)`).
- Async handlers with signature `async void` are not supported.

Using Async in C# Functions with AWS Lambda

If you know your Lambda function will require a long-running process, such as uploading large files to Amazon S3 or reading a large stream of records from DynamoDB, you can take advantage of the `async/await` pattern. When you use this signature, Lambda executes the function synchronously and waits for the function to return a response or for execution to [time out \(p. 35\)](#).

```
public async Task<Response> ProcessS3ImageResizeAsync(SimpleS3Event input)
{
    var response = await client.DoAsyncWork(input);
    return response;
}
```

If you use this pattern, there are some considerations you must take into account:

- AWS Lambda does not support `async void` methods.
- If you create an async Lambda function without implementing the `await` operator, .NET will issue a compiler warning and you will observe unexpected behavior. For example, some `async` actions will execute while others won't. Or some `async` actions won't complete before the function execution is complete.

```
public async Task ProcessS3ImageResizeAsync(SimpleS3Event event) // Compiler warning
{
    client.DoAsyncWork(input);
}
```

- Your Lambda function can include multiple `async` calls, which can be invoked in parallel. You can use the `Task.WhenAll` and `Task.WhenAny` methods to work with multiple tasks. To use the `Task.WhenAll` method, you pass a list of the operations as an array to the method. Note that in the example below, if you neglect to include any operation to the array, that call may return before its operation completes.

```
public async Task DoesNotWaitForAllTasks1()
{
    // In Lambda, Console.WriteLine goes to CloudWatch Logs.
    var task1 = Task.Run(() => Console.WriteLine("Test1"));
    var task2 = Task.Run(() => Console.WriteLine("Test2"));
    var task3 = Task.Run(() => Console.WriteLine("Test3"));

    // Lambda may return before printing "Test2" since we never wait on task2.
    await Task.WhenAll(task1, task3);
}
```

To use the `Task.WhenAny` method, you again pass a list of operations as an array to the method. The call returns as soon as the first operation completes, even if the others are still running.

```
public async Task DoesNotWaitForAllTasks2()
{
```

```
// In Lambda, Console.WriteLine goes to CloudWatch Logs.  
var task1 = Task.Run(() => Console.WriteLine("Test1"));  
var task2 = Task.Run(() => Console.WriteLine("Test2"));  
var task3 = Task.Run(() => Console.WriteLine("Test3"));  
  
// Lambda may return before printing all tests since we're only waiting for one to  
// finish.  
await Task.WhenAny(task1, task2, task3);  
}
```

AWS Lambda Context Object in C#

When Lambda runs your function, it passes a context object to the [handler \(p. 311\)](#). This object provides properties with information about the invocation, function, and execution environment.

Context Properties

- `FunctionName` – The name of the Lambda function.
- `FunctionVersion` – The [version \(p. 47\)](#) of the function.
- `InvokedFunctionArn` – The Amazon Resource Name (ARN) used to invoke the function. Indicates if the invoker specified a version number or alias.
- `MemoryLimitInMB` – The amount of memory configured on the function.
- `AwsRequestId` – The identifier of the invocation request.
- `LogGroupName` – The log group for the function.
- `LogStreamName` – The log stream for the function instance.
- `RemainingTime (TimeSpan)` – The number of milliseconds left before the execution times out.
- `Identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
- `ClientContext` – (mobile apps) Client context provided to the Lambda invoker by the client application.
- `Logger` The [logger object \(p. 315\)](#) for the function.

The following C# code snippet shows a simple handler function that prints some of the context information.

```
public async Task Handler(ILambdaContext context)  
{  
    Console.WriteLine("Function name: " + context.FunctionName);  
    Console.WriteLine("RemainingTime: " + context.RemainingTime);  
    await Task.Delay(TimeSpan.FromSeconds(0.42));  
    Console.WriteLine("RemainingTime after sleep: " + context.RemainingTime);  
}
```

AWS Lambda Function Logging in C#

Your Lambda function comes with a CloudWatch Logs log group, with a log stream for each instance of your function. The runtime sends details about each invocation to the log stream, and relays logs and other output from your function's code.

To output logs from your function code, you can use methods on [the Console class](#), or any logging library that writes to `stdout` or `stderr`. The following example logs the request ID for an invocation.

```
public class ProductService
{
    public async Task<Product> DescribeProduct(DescribeProductRequest request)
    {
        Console.WriteLine("DescribeProduct invoked with Id " + request.Id);
        return await catalogService.DescribeProduct(request.Id);
    }
}
```

Lambda also provides a logger class in the `Amazon.Lambda.Core` library. Use the `Log` method on the `Amazon.Lambda.Core.LambdaLogger` class to write logs.

```
using Amazon.Lambda.Core;

public class ProductService
{
    public async Task<Product> DescribeProduct(DescribeProductRequest request)
    {
        LambdaLogger.Log("DescribeProduct invoked with Id " + request.Id);
        return await catalogService.DescribeProduct(request.Id);
    }
}
```

An instance of this class is also available on [the context object \(p. 315\)](#).

```
public class ProductService
{
    public async Task<Product> DescribeProduct(DescribeProductRequest request,
ILambdaContext context)
    {
        context.Logger.Log("DescribeProduct invoked with Id " + request.Id);
        return await catalogService.DescribeProduct(request.Id);
    }
}
```

The Lambda console shows log output when you test a function on the function configuration page. To view logs for all invocations, use the CloudWatch Logs console.

To view your Lambda function's logs

1. Open the [Logs page of the CloudWatch console](#).
2. Choose the log group for your function (`/aws/lambda/function-name`).
3. Choose the first stream in the list.

Each log stream corresponds to an [instance of your function \(p. 104\)](#). New streams appear when you update your function and when additional instances are created to handle multiple concurrent invocations. To find logs for specific invocations, you can instrument your function with X-Ray and record details about the request and log stream in the trace. For a sample application that correlates logs and traces with X-Ray, see [Error Processor Sample Application for AWS Lambda \(p. 119\)](#).

To get logs for an invocation from the command line, use the `--log-type` option. The response includes a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

```
$ aws lambda invoke --function-name my-function out --log-type Tail
{
    "StatusCode": 200,
    "LogResult": "U1RBULQgUmVxdWVzdElkOia4N2QwNDRiOC1mMTU0LTEzTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
```

```
    "ExecutedVersion": "$LATEST"
}
```

You can use the `base64` utility to decode the logs.

```
$ aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text | base64 -d
START RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d Version: $LATEST
Processing event...
END RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d
REPORT RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d Duration: 29.40 ms      Billed
Duration: 100 ms      Memory Size: 128 MB      Max Memory Used: 19 MB
```

`base64` is available on Linux, macOS, and [Ubuntu on Windows](#). For macOS, the command is `base64 -D`.

Log groups are not deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.

AWS Lambda Function Errors in C#

When an exception occurs in your Lambda function, Lambda will report the exception information back to you. Exceptions can occur in two different places:

- Initialization (Lambda loading your code, validating the handler string, and creating an instance of your class if it is non-static).
- The Lambda function invocation.

The serialized exception information is returned as the payload as a modeled JSON object and outputted to CloudWatch logs.

In the initialization phase, exceptions can be thrown for invalid handler strings, a rule-breaking type or method (see [Lambda Function Handler Restrictions \(p. 313\)](#)), or any other validation method (such as forgetting the serializer attribute and having a POCO as your input or output type). These exceptions are of type `LambdaException`. For example:

```
{
  "errorType": "LambdaException",
  "errorMessage": "Invalid lambda function handler: 'http://this.is.not.a.valid.handler/'.
  The valid format is 'ASSEMBLY::TYPE::METHOD'."
}
```

If your constructor throws an exception, the error type is also of type `LambdaException`, but the exception thrown during construction is provided in the `cause` property, which is itself a modeled exception object:

```
{
  "errorType": "LambdaException",
  "errorMessage": "An exception was thrown when the constructor for type
'Lambdatestfunction.ThrowExceptionInConstructor'
was invoked. Check inner exception for more details.",
  "cause": {
    "errorType": "TargetInvocationException",
    "errorMessage": "Exception has been thrown by the target of an invocation.",
    "stackTrace": [
      "at System.RuntimeTypeHandle.CreateInstance(RuntimeType type, Boolean publicOnly,
Boolean noCheck, Boolean&canBeCached,
```

```
        RuntimeMethodHandleInternal&ctor, Boolean& bNeedSecurityCheck)",
        "at System.RuntimeType.CreateInstanceSlow(Boolean publicOnly, Boolean skipCheckThis,
Boolean fillCache, StackCrawlMark& stackMark)",
        "at System.Activator.CreateInstance(Type type, Boolean nonPublic)",
        "at System.Activator.CreateInstance(Type type)"
    ],
    "cause": {
        "errorType": "ArithmetException",
        "errorMessage": "Sorry, 2 + 2 = 5",
        "stackTrace": [
            "at LambdaExceptionTestFunction.ThrowExceptionInConstructor..ctor()"
        ]
    }
}
```

As the example shows, the inner exceptions are always preserved (as the cause property), and can be deeply nested.

Exceptions can also occur during invocation. In this case, the exception type is preserved and the exception is returned directly as the payload and in the CloudWatch logs. For example:

```
{
    "errorType": "AggregateException",
    "errorMessage": "One or more errors occurred. (An unknown web exception occurred!)",
    "stackTrace": [
        "at System.Threading.Tasks.Task.ThrowIfExceptional(Boolean
includeTaskCanceledExceptions)",
        "at System.Threading.Tasks.Task`1.GetResultCore(Boolean waitCompletionNotification)",
        "at lambda_method(Closure , Stream , Stream , ContextInfo )"
    ],
    "cause": {
        "errorType": "UnknownWebException",
        "errorMessage": "An unknown web exception occurred!",
        "stackTrace": [
            "at LambdaDemo107.LambdaEntryPoint.<GetUriResponse>d__1.MoveNext()",
            "--- End of stack trace from previous location where exception was thrown ---",
            "at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
            "at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
task)",
            "at System.Runtime.CompilerServices.TaskAwaiter`1.GetResult()",  

            "at LambdaDemo107.LambdaEntryPoint.<CheckWebsiteStatus>d__0.MoveNext()"
        ],
        "cause": {
            "errorType": "WebException",
            "errorMessage": "An error occurred while sending the request. SSL peer certificate or
SSH remote key was not OK",
            "stackTrace": [
                "at System.Net.HttpWebRequest.EndGetResponse(IAsyncResult asyncResult)",
                "at System.Threading.Tasks.TaskFactory`1.FromAsyncCoreLogic(IAsyncResult iar,
Func`2 endFunction, Action`1 endAction, Task`1 promise, Boolean requiresSynchronization)",
                "--- End of stack trace from previous location where exception was thrown ---",
                "at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
                "at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
task)",
                "at System.Runtime.CompilerServices.TaskAwaiter`1.GetResult()",  

                "at LambdaDemo107.LambdaEntryPoint.<GetUriResponse>d__1.MoveNext()"
            ],
            "cause": {
                "errorType": "HttpRequestException",
                "errorMessage": "An error occurred while sending the request.",
                "stackTrace": [

```

```
        "at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
        "at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
task)",
        "at System.Net.Http.HttpClient.<FinishSendAsync>d__58.MoveNext()",  
    "---- End of stack trace from previous location where exception was thrown ---",
        "at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
        "at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
task)",
        "at System.Net.HttpWebRequest.<SendRequest>d__63.MoveNext()",  
    "---- End of stack trace from previous location where exception was thrown ---",
        "at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
        "at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
task)",
        "at System.Net.HttpWebRequest.EndGetResponse(IAsyncResult asyncResult)"
    ],
    "cause": {
        "errorType": "CurlException",
        "errorMessage": "SSL peer certificate or SSH remote key was not OK",
        "stackTrace": [
            "at System.Net.Http.CurlHandler.ThrowIfCURLError(CURLcode error)",
            "at
System.Net.Http.CurlHandler.MultiAgent.FinishRequest(StrongToWeakReference`1 easyWrapper,
CURLcode messageResult)"
        ]
    }
}
```

The method in which error information is conveyed depends on the invocation type:

- **RequestResponse** invocation type (that is, synchronous execution): In this case, you get the error message back.

For example, if you invoke a Lambda function using the Lambda console, the RequestResponse is always the invocation type and the console displays the error information returned by AWS Lambda in the **Execution result** section of the console.

- Event invocation type (that is, asynchronous execution): In this case AWS Lambda does not return anything. Instead, it logs the error information in CloudWatch Logs and CloudWatch metrics.

Depending on the event source, AWS Lambda may retry the failed Lambda function. For more information, see [AWS Lambda Retry Behavior \(p. 85\)](#).

Function Error Handling

You can create custom error handling to raise an exception directly from your Lambda function and handle it directly (Retry or Catch) within an AWS Step Functions State Machine. For more information, see [Handling Error Conditions Using a State Machine](#).

Consider a `CreateAccount` state is a task that writes a customer's details to a database using a Lambda function.

- If the task succeeds, an account is created and a welcome email is sent.
 - If a user tries to create an account for a username that already exists, the Lambda function raises an error, causing the state machine to suggest a different username and to retry the account-creation process.

The following code samples demonstrate how to do this. Note that custom errors in C# must extend the `Exception` class.

```
namespace Example {
    public class AccountAlreadyExistsException : Exception {
        public AccountAlreadyExistsException(String message) :
            base(message)
    }
}

namespace Example {
    public class Handler {
        public static void CreateAccount() {
            throw new AccountAlreadyExistsException("Account is in use!");
        }
    }
}
```

You can configure Step Functions to catch the error using a `Catch` rule. Lambda automatically sets the error name to the simple class name of the exception at runtime:

```
{
    "StartAt": "CreateAccount",
    "States": {
        "CreateAccount": {
            "Type": "Task",
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:CreateAccount",
            "Next": "SendWelcomeEmail",
            "Catch": [
                {
                    "ErrorEquals": ["AccountAlreadyExistsException"],
                    "Next": "SuggestAccountName"
                }
            ],
            ...
        }
    }
}
```

At runtime, AWS Step Functions catches the error, [transitioning](#) to the `SuggestAccountName` state as specified in the `Next` transition.

Custom error handling makes it easier to create [serverless](#) applications. This feature integrates with all the languages supported by the Lambda [Programming Model \(p. 31\)](#), allowing you to design your application in the programming languages of your choice, mixing and matching as you go.

To learn more about creating your own serverless applications using AWS Step Functions and AWS Lambda, see [AWS Step Functions](#).

Building Lambda Functions with PowerShell

The following sections explain how [common programming patterns and core concepts](#) apply when you author Lambda function code in PowerShell.

.NET Runtimes

Name	Identifier	Languages	Operating System
.NET Core 2.1	dotnetcore2.1	C# PowerShell Core 6.0	Amazon Linux
.NET Core 1.0	dotnetcore1.0	C#	Amazon Linux

Note that Lambda functions in PowerShell require PowerShell Core 6.0. Windows PowerShell isn't supported.

Before you get started, you must first set up a PowerShell development environment. For instructions on how to do this, see [Setting Up a PowerShell Development Environment \(p. 322\)](#).

To learn about how to use the `AWSLambdaPSCore` module to download sample PowerShell projects from templates, create PowerShell deployment packages, and deploy PowerShell functions to the AWS Cloud, see [Using the AWSLambdaPSCore Module \(p. 322\)](#).

Topics

- [AWS Lambda Deployment Package in PowerShell \(p. 321\)](#)
- [AWS Lambda Function Handler in PowerShell \(p. 324\)](#)
- [AWS Lambda Context Object in PowerShell \(p. 325\)](#)
- [AWS Lambda Function Logging in PowerShell \(p. 326\)](#)
- [AWS Lambda Function Errors in PowerShell \(p. 327\)](#)

AWS Lambda Deployment Package in PowerShell

A PowerShell Lambda deployment package is a ZIP file that contains your PowerShell script, PowerShell modules that are required for your PowerShell script, and the assemblies needed to host PowerShell Core.

AWSLambdaPSCore is a PowerShell module that you can install from the [PowerShell Gallery](#). You use this module to create your PowerShell Lambda deployment package.

You're required to use the `#Requires` statement within your PowerShell scripts to indicate the modules that your scripts depend on. This statement performs two important tasks. 1) It communicates to other developers which modules the script uses, and 2) it identifies the dependent modules that AWS PowerShell tools need to package with the script, as part of the deployment. For more information

about the `#Requires` statement in PowerShell, see [About Requires](#). For more information about PowerShell deployment packages, see [AWS Lambda Deployment Package in PowerShell \(p. 321\)](#).

When your PowerShell Lambda function uses the AWS PowerShell cmdlets, be sure to set a `#Requires` statement that references the `AWSPowerShell.NetCore` module, which supports PowerShell Core—and not the `AWSPowerShell` module, which only supports Windows PowerShell. Also, be sure to use version 3.3.270.0 or newer of `AWSPowerShell.NetCore`, which optimizes the cmdlet import process. If you use an older version, you'll experience longer cold starts. For more information, see [AWS Tools for PowerShell](#).

Before you get started, you must first set up a PowerShell development environment. For instructions on how to do this, see [Setting Up a PowerShell Development Environment \(p. 322\)](#).

Setting Up a PowerShell Development Environment

To set up your development environment for writing PowerShell scripts, do the following:

1. **Install the correct version of PowerShell.** Lambda's support for PowerShell is based on the cross-platform PowerShell Core 6.0 release. This means that you can develop your PowerShell Lambda functions on Windows, Linux, or Mac. If you don't have this version of PowerShell installed, you can find instructions in [Installing PowerShell Core](#).
2. **Install the .NET Core 2.1 SDK.** Because PowerShell Core is built on top of .NET Core, the Lambda support for PowerShell uses the same .NET Core 2.1 Lambda runtime for both .NET Core and PowerShell Lambda functions. The .NET Core 2.1 SDK is used by the new Lambda PowerShell publishing cmdlets to create the Lambda deployment package. The .NET Core 2.1 SDK is available at [.NET downloads](#) on the Microsoft website. Be sure to install the SDK and not the runtime installation.
3. **Install the AWSLambdaPSCore module.** You can install this either from the [PowerShell Gallery](#), or you can install it by using the following PowerShell Core shell command:

```
Install-Module AWSLambdaPSCore -Scope CurrentUser
```

Next Steps

- To learn about writing Lambda functions in PowerShell, see [Building Lambda Functions with PowerShell \(p. 321\)](#).
- To learn about using the AWSLambdaPSCore module to download sample PowerShell projects from templates, creating PowerShell deployment packages, and deploying PowerShell functions to the AWS Cloud, see [Using the AWSLambdaPSCore Module \(p. 322\)](#).

Using the AWSLambdaPSCore Module

The AWSLambdaPSCore module has the following new cmdlets to help author and publish PowerShell Lambda functions.

Cmdlet Name	Description
<code>Get#AWSPowerShellLambdaTemplate</code>	Returns a list of getting started templates.
<code>New#AWSPowerShellLambda</code>	Creates an initial PowerShell script based on a template.
<code>Publish#AWSPowerShellLambda</code>	Publishes a given PowerShell script to Lambda.

Cmdlet Name	Description
New-AWSPowerShellLambdaPackage	Creates a Lambda deployment package that can be used in a CI/CD system for deployment.

To help get started writing and invoking a PowerShell script with Lambda, you can use the `New-AWSPowerShellLambda` cmdlet to create a starter script based on a template. You can use the `Publish-AWSPowerShellLambda` cmdlet to deploy your script to AWS Lambda. Then you can test your script either through the command line or the console.

To create a new PowerShell script, upload it, and test it, follow this procedure:

- View available templates.**

Run the following command to view the list of available templates:

```
PS C:\> Get-AWSPowerShellLambdaTemplate

Template          Description
-----
Basic            Bare bones script
CodeCommitTrigger Script to process AWS CodeCommit Triggers
DetectLabels     Use Amazon Rekognition service to tag image files in Amazon S3
                 with detected labels.
KinesisStreamProcessor Script to process an Amazon Kinesis stream
S3Event          Script to process S3 events
SNSSubscription Script to be subscribed to an Amazon SNS topic
SQSQueueProcessor Script to be subscribed to an Amazon SQS queue
```

Note that new templates are being listed all the time, so this output is just an example.

- Create a basic script.**

Run the following command to create a sample script based on the `Basic` template:

```
New-AWSPowerShellLambda -ScriptName MyFirstPSScript -Template Basic
```

A new file named `MyFirstPSScript.ps1` is created in a new subdirectory of the current directory. The name of the directory is based on the `-ScriptName` parameter. You can use the `-Directory` parameter to choose an alternative directory.

You can see that the new file has the following contents:

```
# PowerShell script file to be executed as a AWS Lambda function.
#
# When executing in Lambda the following variables will be predefined.
#   $LambdaInput - A PSObject that contains the Lambda function input data.
#   $LambdaContext - An Amazon.Lambda.Core.ILambdaContext object that contains
#                   information about the currently running Lambda environment.
#
# The last item in the PowerShell pipeline will be returned as the result of the Lambda
# function.
#
# To include PowerShell modules with your Lambda function, like the
# AWSPowerShell.NetCore module, add a "#Requires" statement
# indicating the module and version.

#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.343.0'}

# Uncomment to send the input to CloudWatch Logs
```

```
# Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
```

3. Edit the sample script.

To see how log messages from your PowerShell script are sent to CloudWatch Logs, uncomment the `Write-Host` line of the sample script.

To demonstrate how you can return data back from your Lambda functions, add a new line at the end of the script with `$PSVersionTable`. This adds the `$PSVersionTable` to the PowerShell pipeline. After the PowerShell script is complete, the last object in the PowerShell pipeline is the return data for the Lambda function. `$PSVersionTable` is a PowerShell global variable that also provides information about the running environment.

After making these changes, the last two lines of the sample script look like this:

```
Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)  
$PSVersionTable
```

4. Publish to AWS Lambda.

After editing the `MyFirstPSScript.ps1` file, change the directory to the script's location. Then run the following command to publish the script to AWS Lambda:

```
Publish-AWSPowerShellLambda -ScriptPath .\MyFirstPSScript.ps1 -Name MyFirstPSScript -  
Region us-east-1
```

Note that the `-Name` parameter specifies the Lambda function name, which appears in the Lambda console. You can use this function to invoke your script manually.

5. Test the Lambda function.

You can test the PowerShell Lambda function that you just published by using the `dotnet` CLI from a command prompt. Use the `lambda invoke-function` command to invoke your function.

```
> dotnet lambda invoke-function MyFirstPSScript
```

For more information about the `dotnet` CLI extension, see [.NET Core CLI \(p. 303\)](#).

AWS Lambda Function Handler in PowerShell

When a Lambda function is invoked, the Lambda handler invokes the PowerShell script.

When the PowerShell script is invoked, the following variables are predefined:

- `$LambdaInput` – A PSObject that contains the input to the handler. This input can be event data (published by an event source) or custom input that you provide, such as a string or any custom data object.
- `$LambdaContext` – An `Amazon.Lambda.Core.ILambdaContext` object that you can use to access information about the current execution—such as the name of the current function, the memory limit, execution time remaining, and logging.

For example, consider the following PowerShell example code.

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.343.0'}  
Write-Host 'Function Name:' $LambdaContext.FunctionName
```

This script returns the `FunctionName` property that's obtained from the `$LambdaContext` variable.

Note

You're required to use the `#Requires` statement within your PowerShell scripts to indicate the modules that your scripts depend on. This statement performs two important tasks. 1) It communicates to other developers which modules the script uses, and 2) it identifies the dependent modules that AWS PowerShell tools need to package with the script, as part of the deployment. For more information about the `#Requires` statement in PowerShell, see [About Requires](#). For more information about PowerShell deployment packages, see [AWS Lambda Deployment Package in PowerShell \(p. 321\)](#).

When your PowerShell Lambda function uses the AWS PowerShell cmdlets, be sure to set a `#Requires` statement that references the `AWSPowerShell.NetCore` module, which supports PowerShell Core—and not the `AWSPowerShell` module, which only supports Windows PowerShell. Also, be sure to use version 3.3.270.0 or newer of `AWSPowerShell.NetCore` which optimizes the cmdlet import process. If you use an older version, you'll experience longer cold starts. For more information, see [AWS Tools for PowerShell](#).

Returning Data

Some Lambda invocations are meant to return data back to their caller. For example, if an invocation was in response to a web request coming from API Gateway, then our Lambda function needs to return back the response. For PowerShell Lambda, the last object that's added to the PowerShell pipeline is the return data from the Lambda invocation. If the object is a string, the data is returned as is. Otherwise the object is converted to JSON by using the `ConvertTo-Json` cmdlet.

For example, consider the following PowerShell statement, which adds `$PSVersionTable` to the PowerShell pipeline:

```
$PSVersionTable
```

After the PowerShell script is finished, the last object in the PowerShell pipeline is the return data for the Lambda function. `$PSVersionTable` is a PowerShell global variable that also provides information about the running environment.

AWS Lambda Context Object in PowerShell

When Lambda runs your function, it passes context information by making a `$LambdaContext` variable available to the [handler \(p. 324\)](#). This variable provides methods and properties with information about the invocation, function, and execution environment.

Context Properties

- `FunctionName` – The name of the Lambda function.
- `FunctionVersion` – The [version \(p. 47\)](#) of the function.
- `InvokedFunctionArn` – The Amazon Resource Name (ARN) used to invoke the function. Indicates if the invoker specified a version number or alias.
- `MemoryLimitInMB` – The amount of memory configured on the function.
- `AwsRequestId` – The identifier of the invocation request.
- `LogGroupName` – The log group for the function.
- `LogStreamName` – The log stream for the function instance.
- `RemainingTime` – The number of milliseconds left before the execution times out.
- `Identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.

- ClientContext – (mobile apps) Client context provided to the Lambda invoker by the client application.
- Logger – The [logger object \(p. 326\)](#) for the function.

The following PowerShell code snippet shows a simple handler function that prints some of the context information.

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.343.0'}
Write-Host 'Function name:' $LambdaContext.FunctionName
Write-Host 'Remaining milliseconds:' $LambdaContext.RemainingTime.TotalMilliseconds
Write-Host 'Log group name:' $LambdaContext.LogGroupName
Write-Host 'Log stream name:' $LambdaContext.LogStreamName
```

AWS Lambda Function Logging in PowerShell

Your Lambda function comes with a CloudWatch Logs log group, with a log stream for each instance of your function. The runtime sends details about each invocation to the log stream, and relays logs and other output from your function's code.

To output logs from your function code, you can use cmdlets on [Microsoft.PowerShell.Utility](#), or any logging module that writes to `stdout` or `stderr`. The following example uses `Write-Host`.

```
Write-Host 'Event received.'
```

The Lambda console shows log output when you test a function on the function configuration page. To view logs for all invocations, use the CloudWatch Logs console.

To view your Lambda function's logs

1. Open the [Logs page of the CloudWatch console](#).
2. Choose the log group for your function (`/aws/lambda/function-name`).
3. Choose the first stream in the list.

Each log stream corresponds to an [instance of your function \(p. 104\)](#). New streams appear when you update your function and when additional instances are created to handle multiple concurrent invocations. To find logs for specific invocations, you can instrument your function with X-Ray and record details about the request and log stream in the trace. For a sample application that correlates logs and traces with X-Ray, see [Error Processor Sample Application for AWS Lambda \(p. 119\)](#).

To get logs for an invocation from the command line, use the `--log-type` option. The response includes a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

```
$ aws lambda invoke --function-name my-function out --log-type Tail
{
    "StatusCode": 200,
    "LogResult":
    "U1RBULQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTEzTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

You can use the `base64` utility to decode the logs.

```
$ aws lambda invoke --function-name my-function out --log-type Tail \
```

```
--query 'LogResult' --output text | base64 -d
START RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d Version: $LATEST
Processing event...
END RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d
REPORT RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d Duration: 29.40 ms      Billed
Duration: 100 ms          Memory Size: 128 MB      Max Memory Used: 19 MB
```

`base64` is available on Linux, macOS, and [Ubuntu on Windows](#). For macOS, the command is `base64 -D`.

Log groups are not deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.

AWS Lambda Function Errors in PowerShell

If your Lambda function has a terminating error, AWS Lambda recognizes the failure, serializes the error information into JSON, and returns it.

Consider the following PowerShell script example statement:

```
throw 'The Account is not found'
```

When you invoke this Lambda function, it throws a terminating error, and AWS Lambda returns the following error message:

```
{
  "errorMessage": "The Account is not found",
  "errorType": "RuntimeException"
}
```

Note the `errorType` is `RuntimeException`, which is the default exception thrown by PowerShell. You can use custom error types by throwing the error like this:

```
throw @{'Exception'='AccountNotFound';'Message'='The Account is not found'}
```

The error message is serialized with `errorType` set to `AccountNotFound`:

```
{
  "errorMessage": "The Account is not found",
  "errorType": "AccountNotFound"
}
```

If you don't need an error message, you can throw a string in the format of an error code. The error code format requires that the string starts with a character and only contain letters and digits afterwards, with no spaces or symbols.

For example, if your Lambda function contains the following:

```
throw 'AccountNotFound'
```

The error is serialized like this:

```
{
  "errorMessage": "AccountNotFound",
  "errorType": "AccountNotFound"
```

}

Function Error Handling

You can use a custom `errorType` in your Lambda function and handle function errors directly (Retry or Catch) within an AWS Step Functions State Machine. For more information, see [Handling Error Conditions Using a State Machine](#).

Custom error handling makes it easier to create [serverless](#) applications. This feature integrates with all the languages that are supported by the Lambda [Programming Model \(p. 31\)](#). This allows you to design your application in the programming languages of your choice, mixing and matching as you go.

To learn more about creating your own serverless applications using AWS Step Functions and AWS Lambda, see [AWS Step Functions](#).

Building Lambda Functions with Ruby

You can run Ruby code in AWS Lambda. Lambda provides a [runtime \(p. 102\)](#) for Ruby that executes your code to process events. Your code runs in an environment that includes the AWS SDK for Ruby, with credentials from an AWS Identity and Access Management (IAM) role that you manage.

Lambda supports the following Ruby runtimes:

Ruby Runtimes

Name	Identifier	Operating System
Ruby 2.5	<code>ruby2.5</code>	Amazon Linux

If you don't already have an execution role for function development, create one.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – Lambda.**
 - **Permissions – AWSLambdaBasicExecutionRole.**
 - **Role name – lambda-role.**

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

You can add permissions to the role later, or swap it out for a different role that's specific to a single function.

To create a Ruby function

1. Open the [Lambda console](#).
2. Choose **Create function**.
3. Configure the following settings:
 - **Name – ruby-function.**
 - **Runtime – Ruby 2.5.**
 - **Role – Choose an existing role.**
 - **Existing role – lambda-role.**
4. Choose **Create function**.
5. To configure a test event, choose **Test**.

6. For **Event name**, enter **test**.
7. Choose **Create**.
8. To execute the function, choose **Test**.

The console creates a Lambda function with a single source file named `lambda_function.rb`. You can edit this file and add more files in the built-in [code editor \(p. 24\)](#). Choose **Save** to save your changes, and choose **Test** to run your code.

Note

The Lambda console uses AWS Cloud9 to provide an integrated development environment in the browser. You can also use AWS Cloud9 to develop Lambda functions in your own environment. See [Working with AWS Lambda Functions](#) in the AWS Cloud9 user guide for more information.

`lambda_function.rb` defines a function named `lambda_handler` that takes an event object and a context object. This is the [handler function \(p. 330\)](#) that Lambda calls when the function is invoked. The Ruby function runtime gets invocation events from Lambda and passes them to the handler.

Each time you save your function code, the Lambda console creates a deployment package, which is a ZIP archive that contains your function code. As your function development progresses, you will want to store your function code in source control, add libraries, and automate deployments. Start by [creating a deployment package \(p. 331\)](#) and updating your code at the command line.

The function runtime passes a context object to the handler, in addition to the invocation event. The [context object \(p. 333\)](#) contains additional information about the invocation, the function, and the execution environment. Further information is available from environment variables.

Your Lambda function comes with a CloudWatch Logs log group. The function runtime sends details about each invocation to CloudWatch Logs, and relays any [logs that your function outputs \(p. 334\)](#) during invocation. If your function [returns an error \(p. 335\)](#), Lambda formats the error and returns it to the invoker.

Topics

- [AWS Lambda Function Handler in Ruby \(p. 330\)](#)
- [AWS Lambda Deployment Package in Ruby \(p. 331\)](#)
- [AWS Lambda Context Object in Ruby \(p. 333\)](#)
- [AWS Lambda Function Logging in Ruby \(p. 334\)](#)
- [AWS Lambda Function Errors in Ruby \(p. 335\)](#)

AWS Lambda Function Handler in Ruby

Your Lambda function's handler is the method that Lambda calls when your function is invoked. In the following example, the file `function.rb` defines a handler method named `handler`. The handler function takes two objects as input and returns a JSON document.

Example `function.rb`

```
require 'json'

def handler(event:, context:)
    { event: JSON.generate(event), context: JSON.generate(context.inspect) }
end
```

In your function configuration, the `handler` setting tells Lambda where to find the handler. For the preceding example, the correct value for this setting is `function.handler`. It includes two names separated by a dot: the name of the file and the name of the handler method.

You can also define your handler method in a class. The following example defines a handler method named `process` on a class named `Handler` in a module named `LambdaFunctions`.

Example source.rb

```
module LambdaFunctions
  class Handler
    def self.process(event:, context:)
      "Hello!"
    end
  end
end
```

In this case, the handler setting is `source.LambdaFunctions::Handler.process`.

The two objects that the handler accepts are the invocation event and context. The event is a Ruby object that contains the payload that's provided by the invoker. If the payload is a JSON document, the event object is a Ruby hash. Otherwise, it's a string. The [context object \(p. 333\)](#) has methods and properties that provide information about the invocation, the function, and the execution environment.

The function handler is executed every time your Lambda function is invoked. Static code outside of the handler is executed once per instance of the function. If your handler uses resources like SDK clients and database connections, you can create them outside of the handler method to reuse them for multiple invocations.

Each instance of your function can process multiple invocation events, but it only processes one event at a time. The number of instances processing an event at any given time is your function's *concurrency*. For more information about the Lambda execution context, see [AWS Lambda Execution Context \(p. 104\)](#).

AWS Lambda Deployment Package in Ruby

A deployment package is a ZIP archive that contains your function code and dependencies. You need to create a deployment package if you use the Lambda API to manage functions, or if your code uses libraries other than the AWS SDK. Other libraries and dependencies need to be included in the deployment package. You can upload the package directly to Lambda, or you can use an Amazon S3 bucket, and then upload it to Lambda.

If you use the Lambda [console editor \(p. 24\)](#) to author your function, the console manages the deployment package. You can use this method as long as you don't need to add any libraries. You can also use it to update a function that already has libraries in the deployment package, as long as the total size doesn't exceed 3 MB.

Sections

- [Updating a Function with No Dependencies \(p. 331\)](#)
- [Updating a Function with Additional Dependencies \(p. 332\)](#)

Updating a Function with No Dependencies

To create or update a function with the Lambda API, create an archive that contains your function code and upload it with the AWS CLI.

To update a Ruby function with no dependencies

1. Create a ZIP archive.

```
~/my-function$ zip function.zip function.rb
```

2. Use the update-function-code command to upload the package.

```
~/my-function$ aws lambda update-function-code --function-name ruby25 --zip-file fileb://function.zip
{
    "FunctionName": "ruby25",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:ruby25",
    "Runtime": "ruby2.5",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Handler": "function.handler",
    "CodeSize": 300,
    "Description": "",
    "Timeout": 3,
    "MemorySize": 128,
    "LastModified": "2018-11-23T21:00:10.248+0000",
    "CodeSha256": "Qf0haXm3JtGTmcDbjMc1I2di6YFMi9niEuiYonYptAk=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "Active"
    },
    "RevisionId": "d1e983e3-ca8e-434b-8dc1-7add83d72ebd"
}
```

Updating a Function with Additional Dependencies

If your function depends on libraries other than the AWS SDK for Ruby, install them to a local directory with [Bundler](#), and include them in your deployment package.

To update a Ruby function with dependencies

1. Install libraries in the vendor directory with the bundle command.

```
~/my-function$ bundle install --path vendor/bundle
Fetching gem metadata from https://rubygems.org/.....
Resolving dependencies...
Fetching aws-eventstream 1.0.1
Installing aws-eventstream 1.0.1
...
```

The --path installs the gems in the project directory instead of the system location, and sets this as the default path for future installations. To later install gems globally, use the --system option.

2. Create a ZIP archive.

```
package$ zip -r function.zip function.rb vendor
adding: function.rb (deflated 37%)
adding: vendor/ (stored 0%)
adding: vendor/bundle/ (stored 0%)
adding: vendor/bundle/ruby/ (stored 0%)
adding: vendor/bundle/ruby/2.5.0/ (stored 0%)
adding: vendor/bundle/ruby/2.5.0/build_info/ (stored 0%)
adding: vendor/bundle/ruby/2.5.0/cache/ (stored 0%)
adding: vendor/bundle/ruby/2.5.0/cache/aws-eventstream-1.0.1.gem (deflated 36%)
```

...

3. Update the function code.

```
~/my-function$ aws lambda update-function-code --function-name ruby25 --zip-file
fileb://function.zip
{
    "FunctionName": "ruby25",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:ruby25",
    "Runtime": "ruby2.5",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Handler": "function.handler",
    "CodeSize": 998918,
    "Description": "",
    "Timeout": 3,
    "MemorySize": 128,
    "LastModified": "2018-11-20T20:51:35.871+0000",
    "CodeSha256": "fJ3TxYnFosnnpN483dz9/rTzcXrbOiuu4iOZx34nXZI=",
    "Version": "$LATEST",
    "VpcConfig": {
        "SubnetIds": [],
        "SecurityGroupIds": [],
        "VpcId": ""
    },
    "TracingConfig": {
        "Mode": "Active"
    },
    "RevisionId": "9ca7c45b-bcda-4e51-ab5f-7c42fa916e39"
}
```

AWS Lambda Context Object in Ruby

When Lambda runs your function, it passes a context object to the [handler \(p. 330\)](#). This object provides methods and properties that provide information about the invocation, function, and execution environment.

Context Methods

- `get_remaining_time_in_millis` – Returns the number of milliseconds left before the execution times out.

Context Properties

- `function_name` – The name of the Lambda function.
- `function_version` – The [version \(p. 47\)](#) of the function.
- `invoked_function_arn` – The Amazon Resource Name (ARN) used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memory_limit_in_mb` – The amount of memory configured on the function.
- `aws_request_id` – The identifier of the invocation request.
- `log_group_name` – The log group for the function.
- `log_stream_name` – The log stream for the function instance.
- `deadline_ms` – The date that the execution times out, in Unix time milliseconds.
- `identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.

- `client_context`—(mobile apps) Client context provided to the Lambda invoker by the client application.

AWS Lambda Function Logging in Ruby

Your Lambda function comes with a CloudWatch Logs log group, with a log stream for each instance of your function. The runtime sends details about each invocation to the log stream, and relays logs and other output from your function's code.

To output logs from your function code, you can use `puts` statements, or any logging library that writes to `stdout` or `stderr`. The following example logs the values of environment variables and the event object.

Example `lambda_function.rb`

```
# lambda_function.rb

def handler(event:, context:)
    puts "## ENVIRONMENT VARIABLES"
    puts ENV.to_a
    puts "## EVENT"
    puts event.to_a
end
```

The Lambda console shows log output when you test a function on the function configuration page. To view logs for all invocations, use the CloudWatch Logs console.

To view your Lambda function's logs

1. Open the [Logs page of the CloudWatch console](#).
2. Choose the log group for your function (`/aws/lambda/function-name`).
3. Choose the first stream in the list.

Each log stream corresponds to an [instance of your function \(p. 104\)](#). New streams appear when you update your function and when additional instances are created to handle multiple concurrent invocations. To find logs for specific invocations, you can instrument your function with X-Ray and record details about the request and log stream in the trace. For a sample application that correlates logs and traces with X-Ray, see [Error Processor Sample Application for AWS Lambda \(p. 119\)](#).

To get logs for an invocation from the command line, use the `--log-type` option. The response includes a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

```
$ aws lambda invoke --function-name my-function out --log-type Tail
{
    "StatusCode": 200,
    "LogResult": "U1RBUL0gUmVxdWVzdElkOiaA4N2QwNDRiOC1mMTU0LTEzTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

You can use the `base64` utility to decode the logs.

```
$ aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text | base64 -d
START RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d Version: $LATEST
```

```
Processing event...
END RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d
REPORT RequestId: 8e827ab1-f155-11e8-b06d-018ab046158d Duration: 29.40 ms      Billed
Duration: 100 ms          Memory Size: 128 MB      Max Memory Used: 19 MB
```

base64 is available on Linux, macOS, and [Ubuntu on Windows](#). For macOS, the command is `base64 -D`.

Log groups are not deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.

AWS Lambda Function Errors in Ruby

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for a synchronous invocation, in the output.

Example `function.rb`

```
def handler(event:, context:)
    puts "Processing event..."
    [1, 2, 3].first("two")
    "Success"
end
```

This code results in a type error. Lambda catches the error and generates a JSON document with fields for the error message, the type, and the stack trace.

```
{
  "errorMessage": "no implicit conversion of String into Integer",
  "errorType": "Function<TypeError>",
  "stackTrace": [
    "/var/task/function.rb:3:in `first'",
    "/var/task/function.rb:3:in `handler'"
  ]
}
```

When you invoke the function from the command line, the status code doesn't change, but the response includes a `FunctionError` field, and the output includes the error document.

```
$ aws lambda invoke --function-name ruby-function out
{
  "StatusCode": 200,
  "FunctionError": "UncaughtException",
  "ExecutedVersion": "$LATEST"
}
$ cat out
{"errorMessage": "no implicit conversion of String into Integer", "errorType": "Function<TypeError>", "stackTrace": ["/var/task/function.rb:3:in `first'", "/var/task/function.rb:3:in `handler'"]}
```

To view the error in the error log, use the `--log-type` option and decode the base64 string in the response.

```
$ aws lambda invoke --function-name ruby-function out --log-type Tail \
--query 'LogResult' --output text | base64 -d
START RequestId: 5ce6a15a-f156-11e8-b8aa-25371a5ca2a3 Version: $LATEST
Processing event...
Error raised from handler method
```

```
{  
    "errorMessage": "no implicit conversion of String into Integer",  
    "errorType": "Function<TypeError>",  
    "stackTrace": [  
        "/var/task/function.rb:3:in `first'",  
        "/var/task/function.rb:3:in `handler'"  
    ]  
}  
END RequestId: 5ce6a15a-f156-11e8-b8aa-25371a5ca2a3  
REPORT RequestId: 5ce6a15a-f156-11e8-b8aa-25371a5ca2a3 Duration: 22.74 ms      Billed  
Duration: 100 ms          Memory Size: 128 MB     Max Memory Used: 18 MB
```

For more information, see [AWS Lambda Function Logging in Ruby \(p. 334\)](#).

API Reference

This section contains the AWS Lambda API Reference documentation. When making the API calls, you will need to authenticate your request by providing a signature. AWS Lambda supports signature version 4. For more information, see [Signature Version 4 Signing Process](#) in the *Amazon Web Services General Reference*.

For an overview of the service, see [What Is AWS Lambda? \(p. 1\)](#).

You can use the AWS CLI to explore the AWS Lambda API. This guide provides several tutorials that use the AWS CLI.

Topics

- [Actions \(p. 337\)](#)
- [Data Types \(p. 470\)](#)

Actions

The following actions are supported:

- [AddLayerVersionPermission \(p. 339\)](#)
- [AddPermission \(p. 343\)](#)
- [CreateAlias \(p. 347\)](#)
- [CreateEventSourceMapping \(p. 351\)](#)
- [CreateFunction \(p. 355\)](#)
- [DeleteAlias \(p. 363\)](#)
- [DeleteEventSourceMapping \(p. 365\)](#)
- [DeleteFunction \(p. 368\)](#)
- [DeleteFunctionConcurrency \(p. 370\)](#)
- [DeleteLayerVersion \(p. 372\)](#)
- [GetAccountSettings \(p. 374\)](#)
- [GetAlias \(p. 376\)](#)
- [GetEventSourceMapping \(p. 379\)](#)
- [GetFunction \(p. 382\)](#)
- [GetFunctionConfiguration \(p. 385\)](#)
- [GetLayerVersion \(p. 390\)](#)
- [GetLayerVersionByArn \(p. 393\)](#)
- [GetLayerVersionPolicy \(p. 396\)](#)
- [GetPolicy \(p. 398\)](#)
- [Invoke \(p. 400\)](#)
- [InvokeAsync \(p. 405\)](#)
- [ListAliases \(p. 407\)](#)
- [ListEventSourceMappings \(p. 410\)](#)
- [ListFunctions \(p. 413\)](#)
- [ListLayers \(p. 416\)](#)

- [ListLayerVersions \(p. 418\)](#)
- [ListTags \(p. 421\)](#)
- [ListVersionsByFunction \(p. 423\)](#)
- [PublishLayerVersion \(p. 426\)](#)
- [PublishVersion \(p. 430\)](#)
- [PutFunctionConcurrency \(p. 436\)](#)
- [RemoveLayerVersionPermission \(p. 439\)](#)
- [RemovePermission \(p. 441\)](#)
- [TagResource \(p. 444\)](#)
- [UntagResource \(p. 446\)](#)
- [UpdateAlias \(p. 448\)](#)
- [UpdateEventSourceMapping \(p. 452\)](#)
- [UpdateFunctionCode \(p. 456\)](#)
- [UpdateFunctionConfiguration \(p. 463\)](#)

AddLayerVersionPermission

Adds permissions to the resource-based policy of a version of an [AWS Lambda layer](#). Use this action to grant layer usage permission to other accounts. You can grant permission to a single account, all AWS accounts, or all accounts in an organization.

To revoke permission, call [RemoveLayerVersionPermission \(p. 439\)](#) with the statement ID that you specified when you added it.

Request Syntax

```
POST /2018-10-31/layers/LayerName/versions/VersionNumber/policy?RevisionId=RevisionId
HTTP/1.1
Content-type: application/json

{
    "Action": "string",
    "OrganizationId": "string",
    "Principal": "string",
    "StatementId": "string"
}
```

URI Request Parameters

The request requires the following URI parameters.

[LayerName \(p. 339\)](#)

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+)|[a-zA-Z0-9-_]+

[RevisionId \(p. 339\)](#)

Only update the policy if the revision ID matches the ID specified. Use this option to avoid modifying a policy that has changed since you last read it.

[VersionNumber \(p. 339\)](#)

The version number.

Request Body

The request accepts the following data in JSON format.

[Action \(p. 339\)](#)

The API action that grants access to the layer. For example, lambda:GetLayerVersion.

Type: String

Pattern: lambda:GetLayerVersion

Required: Yes

[OrganizationId \(p. 339\)](#)

With the principal set to *, grant permission to all accounts in the specified organization.

Type: String

Pattern: o-[a-zA-Z0-9]{10,32}

Required: No

[Principal \(p. 339\)](#)

An account ID, or * to grant permission to all AWS accounts.

Type: String

Pattern: \d{12}|*|arn:(aws[a-zA-Z-]*):iam::\d{12}:root

Required: Yes

[StatementId \(p. 339\)](#)

An identifier that distinguishes the policy from others on the same layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 100.

Pattern: ([a-zA-Z0-9-_]+)

Required: Yes

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
    "RevisionId": "string",
    "Statement": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

[RevisionId \(p. 340\)](#)

A unique identifier for the current revision of the policy.

Type: String

[Statement \(p. 340\)](#)

The permission statement.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

PolicyLengthExceededException

The permissions policy for the resource is too large. [Learn more](#)

HTTP Status Code: 400

PreconditionFailedException

The `RevisionId` provided does not match the latest `RevisionId` for the Lambda function or alias. Call the `GetFunction` or the `GetAlias` API to retrieve the latest `RevisionId` for your resource.

HTTP Status Code: 412

ResourceConflictException

The resource already exists.

HTTP Status Code: 409

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)

- [AWS SDK for Ruby V2](#)

AddPermission

Grants an AWS service or another account permission to use a function. You can apply the policy at the function level, or specify a qualifier to restrict access to a single version or alias. If you use a qualifier, the invoker must use the full Amazon Resource Name (ARN) of that version or alias to invoke the function.

To grant permission to another account, specify the account ID as the `Principal`. For AWS services, the principal is a domain-style identifier defined by the service, like `s3.amazonaws.com` or `sns.amazonaws.com`. For AWS services, you can also specify the ARN or owning account of the associated resource as the `SourceArn` or `SourceAccount`. If you grant permission to a service principal without specifying the source, other accounts could potentially configure resources in their account to invoke your Lambda function.

This action adds a statement to a resource-based permission policy for the function. For more information about function policies, see [Lambda Function Policies](#).

Request Syntax

```
POST /2015-03-31/functions/FunctionName/policy?Qualifier=Qualifier HTTP/1.1
Content-type: application/json

{
  "Action": "string",
  "EventSourceToken": "string",
  "Principal": "string",
  "RevisionId": "string",
  "SourceAccount": "string",
  "SourceArn": "string",
  "StatementId": "string"
}
```

URI Request Parameters

The request requires the following URI parameters.

[FunctionName](#) (p. 343)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** - `my-function` (name-only), `my-function:v1` (with alias).
- **Function ARN** - `arn:aws:lambda:us-west-2:123456789012:function:my-function`.
- **Partial ARN** - `123456789012:function:my-function`.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[Qualifier](#) (p. 343)

Specify a version or alias to add permissions to a published version of the function.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (| [a-zA-Z0-9\$_-]+)

Request Body

The request accepts the following data in JSON format.

Action (p. 343)

The action that the principal can use on the function. For example, `lambda:InvokeFunction` or `lambda:GetFunction`.

Type: String

Pattern: (`lambda:[*]` | `lambda:[a-zA-Z]+[*]`)

Required: Yes

EventSourceToken (p. 343)

For Alexa Smart Home functions, a token that must be supplied by the invoker.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Pattern: [a-zA-Z0-9._\-_]+

Required: No

Principal (p. 343)

The AWS service or account that invokes the function. If you specify a service, use `SourceArn` or `SourceAccount` to limit who can invoke the function through that service.

Type: String

Pattern: .*

Required: Yes

RevisionId (p. 343)

Only update the policy if the revision ID matches the ID that's specified. Use this option to avoid modifying a policy that has changed since you last read it.

Type: String

Required: No

SourceAccount (p. 343)

For AWS services, the ID of the account that owns the resource. Use this instead of `SourceArn` to grant permission to resources that are owned by another account (for example, all of an account's Amazon S3 buckets). Or use it together with `SourceArn` to ensure that the resource is owned by the specified account. For example, an Amazon S3 bucket could be deleted by its owner and recreated by another account.

Type: String

Pattern: \d{12}

Required: No

[SourceArn \(p. 343\)](#)

For AWS services, the ARN of the AWS resource that invokes the function. For example, an Amazon S3 bucket or Amazon SNS topic.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+:(a-z){2}(-gov)?-[a-zA-Z]+\d{1})?:(\d{12})?:(.*)

Required: No

[StatementId \(p. 343\)](#)

A statement identifier that differentiates the statement from others in the same policy.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 100.

Pattern: ([a-zA-Z0-9-_]+)

Required: Yes

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
    "Statement": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

[Statement \(p. 345\)](#)

The permission statement that's added to the function policy.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

PolicyLengthExceededException

The permissions policy for the resource is too large. [Learn more](#)

HTTP Status Code: 400

PreconditionFailedException

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceConflictException

The resource already exists.

HTTP Status Code: 409

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

CreateAlias

Creates an [alias](#) for a Lambda function version. Use aliases to provide clients with a function identifier that you can update to invoke a different version.

You can also map an alias to split invocation requests between two versions. Use the `RoutingConfig` parameter to specify a second version and the percentage of invocation requests that it receives.

Request Syntax

```
POST /2015-03-31/functions/FunctionName/aliases HTTP/1.1
Content-type: application/json

{
  "Description": "string",
  "FunctionVersion": "string",
  "Name": "string",
  "RoutingConfig": {
    "AdditionalVersionWeights": {
      "string": number
    }
  }
}
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 347)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}):?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Request Body

The request accepts the following data in JSON format.

Description (p. 347)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

[FunctionVersion \(p. 347\)](#)

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST|[0-9]+)

Required: Yes

[Name \(p. 347\)](#)

The name of the alias.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (?![0-9]+\$)([a-zA-Z0-9-_]+)

Required: Yes

[RoutingConfig \(p. 347\)](#)

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 475\)](#) object

Required: No

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
    "AliasArn": "string",
    "Description": "string",
    "FunctionVersion": "string",
    "Name": "string",
    "RevisionId": "string",
    "RoutingConfig": {
        "AdditionalVersionWeights": {
            "string" : number
        }
    }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

[AliasArn \(p. 348\)](#)

The Amazon Resource Name (ARN) of the alias.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Description (p. 348)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

FunctionVersion (p. 348)

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST|[0-9]+)

Name (p. 348)

The name of the alias.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (?![0-9]+\$)([a-zA-Z0-9-_]+)

RevisionId (p. 348)

A unique identifier that changes when you update the alias.

Type: String

RoutingConfig (p. 348)

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 475\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceConflictException

The resource already exists.

HTTP Status Code: 409

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

CreateEventSourceMapping

Creates a mapping between an event source and an AWS Lambda function. Lambda reads items from the event source and triggers the function.

For details about each event source type, see the following topics.

- [Using AWS Lambda with Amazon Kinesis](#)
- [Using AWS Lambda with Amazon SQS](#)
- [Using AWS Lambda with Amazon DynamoDB](#)

Request Syntax

```
POST /2015-03-31/event-source-mappings/ HTTP/1.1
Content-type: application/json

{
    "BatchSize": number,
    "Enabled": boolean,
    "EventSourceArn": "string",
    "FunctionName": "string",
    "StartingPosition": "string",
    "StartingPositionTimestamp": number
}
```

URI Request Parameters

The request does not use any URI parameters.

Request Body

The request accepts the following data in JSON format.

BatchSize (p. 351)

The maximum number of items to retrieve in a single batch.

- **Amazon Kinesis** - Default 100. Max 10,000.
- **Amazon DynamoDB Streams** - Default 100. Max 1,000.
- **Amazon Simple Queue Service** - Default 10. Max 10.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

Required: No

Enabled (p. 351)

Disables the event source mapping to pause polling and invocation.

Type: Boolean

Required: No

EventSourceArn (p. 351)

The Amazon Resource Name (ARN) of the event source.

- **Amazon Kinesis** - The ARN of the data stream or a stream consumer.
- **Amazon DynamoDB Streams** - The ARN of the stream.
- **Amazon Simple Queue Service** - The ARN of the queue.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+:(a-z){2}(-gov)?-[a-zA-Z]+\d{1}):(\d{12}):(.*)

Required: Yes

[FunctionName \(p. 351\)](#)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Version or Alias ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction:PROD.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it's limited to 64 characters in length.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-zA-Z]{2}(-gov)?-[a-zA-Z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[StartingPosition \(p. 351\)](#)

The position in a stream from which to start reading. Required for Amazon Kinesis and Amazon DynamoDB Streams sources. AT_TIMESTAMP is only supported for Amazon Kinesis streams.

Type: String

Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

Required: No

[StartingPositionTimestamp \(p. 351\)](#)

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds.

Type: Timestamp

Required: No

Response Syntax

```
HTTP/1.1 202
Content-type: application/json
```

```
{  
    "BatchSize": number,  
    "EventSourceArn": "string",  
    "FunctionArn": "string",  
    "LastModified": number,  
    "LastProcessingResult": "string",  
    "State": "string",  
    "StateTransitionReason": "string",  
    "UUID": "string"  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 202 response.

The following data is returned in JSON format by the service.

[BatchSize \(p. 352\)](#)

The maximum number of items to retrieve in a single batch.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

[EventSourceArn \(p. 352\)](#)

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+:([a-z]{2}(-gov)?-[a-z]+\-\d{1})?:(\d{12})?:(.*)

[FunctionArn \(p. 352\)](#)

The ARN of the Lambda function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+\-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$\\$LATEST|[a-zA-Z0-9-_]+))?

[LastModified \(p. 352\)](#)

The date that the event source mapping was last updated, in Unix time seconds.

Type: Timestamp

[LastProcessingResult \(p. 352\)](#)

The result of the last AWS Lambda invocation of your Lambda function.

Type: String

[State \(p. 352\)](#)

The state of the event source mapping. It can be one of the following: Creating, Enabling, Enabled, Disabling, Disabled, Updating, or Deleting.

Type: String

[StateTransitionReason \(p. 352\)](#)

The cause of the last state change, either User initiated or Lambda initiated.

Type: String

UUID (p. 352)

The identifier of the event source mapping.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceConflictException

The resource already exists.

HTTP Status Code: 409

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

CreateFunction

Creates a Lambda function. To create a function, you need a [deployment package](#) and an [execution role](#). The deployment package contains your function code. The execution role grants the function permission to use AWS services, such as Amazon CloudWatch Logs for log streaming and AWS X-Ray for request tracing.

A function has an unpublished version, and can have published versions and aliases. The unpublished version changes when you update your function's code and configuration. A published version is a snapshot of your function code and configuration that can't be changed. An alias is a named resource that maps to a version, and can be changed to map to a different version. Use the `Publish` parameter to create version 1 of your function from its initial configuration.

The other parameters let you configure version-specific and function-level settings. You can modify version-specific settings later with [UpdateFunctionConfiguration \(p. 463\)](#). Function-level settings apply to both the unpublished and published versions of the function, and include tags ([TagResource \(p. 444\)](#)) and per-function concurrency limits ([PutFunctionConcurrency \(p. 436\)](#)).

If another account or an AWS service invokes your function, use [AddPermission \(p. 343\)](#) to grant permission by creating a resource-based IAM policy. You can grant permissions at the function level, on a version, or on an alias.

To invoke your function directly, use [Invoke \(p. 400\)](#). To invoke your function in response to events in other AWS services, create an event source mapping ([CreateEventSourceMapping \(p. 351\)](#)), or configure a function trigger in the other service. For more information, see [Invoking Functions](#).

Request Syntax

```
POST /2015-03-31/functions HTTP/1.1
Content-type: application/json

{
  "Code": {
    "S3Bucket": "string",
    "S3Key": "string",
    "S3ObjectVersion": "string",
    "ZipFile": blob
  },
  "DeadLetterConfig": {
    "TargetArn": "string"
  },
  "Description": "string",
  "Environment": {
    "Variables": {
      "string" : "string"
    }
  },
  "FunctionName": "string",
  "Handler": "string",
  "KMSKeyArn": "string",
  "Layers": [ "string" ],
  "MemorySize": number,
  "Publish": boolean,
  "Role": "string",
  "Runtime": "string",
  "Tags": {
    "string" : "string"
  },
  "Timeout": number,
  "TracingConfig": {
    "Mode": "string"
  }
}
```

```
    },
    "VpcConfig": {
        "SecurityGroupIds": [ "string" ],
        "SubnetIds": [ "string" ]
    }
}
```

URI Request Parameters

The request does not use any URI parameters.

Request Body

The request accepts the following data in JSON format.

[Code \(p. 355\)](#)

The code for the function.

Type: [FunctionCode \(p. 483\)](#) object

Required: Yes

[DeadLetterConfig \(p. 355\)](#)

A dead letter queue configuration that specifies the queue or topic where Lambda sends asynchronous events when they fail processing. For more information, see [Dead Letter Queues](#).

Type: [DeadLetterConfig \(p. 477\)](#) object

Required: No

[Description \(p. 355\)](#)

A description of the function.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

[Environment \(p. 355\)](#)

Environment variables that are accessible from function code during execution.

Type: [Environment \(p. 478\)](#) object

Required: No

[FunctionName \(p. 355\)](#)

The name of the Lambda function.

Name formats

- **Function name** - my-function.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: Yes

[Handler \(p. 355\)](#)

The name of the method within your code that Lambda calls to execute your function. The format includes the file name. It can also include namespaces and other qualifiers, depending on the runtime. For more information, see [Programming Model](#).

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

Required: Yes

[KMSKeyArn \(p. 355\)](#)

The ARN of the AWS Key Management Service (AWS KMS) key that's used to encrypt your function's environment variables. If it's not provided, AWS Lambda uses a default service key.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-.]+:[^.]+|())`

Required: No

[Layers \(p. 355\)](#)

A list of [function layers](#) to add to the function's execution environment. Specify each layer by its ARN, including the version.

Type: Array of strings

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+`

Required: No

[MemorySize \(p. 355\)](#)

The amount of memory that your function has access to. Increasing the function's memory also increases its CPU allocation. The default value is 128 MB. The value must be a multiple of 64 MB.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 3008.

Required: No

[Publish \(p. 355\)](#)

Set to true to publish the first version of the function during creation.

Type: Boolean

Required: No

[Role \(p. 355\)](#)

The Amazon Resource Name (ARN) of the function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/[a-zA-Z_0-9+=,.@\-_/.]+`

Required: Yes

[Runtime \(p. 355\)](#)

The identifier of the function's [runtime](#).

Type: String

Valid Values: `nodejs8.10` | `nodejs10.x` | `java8` | `python2.7` | `python3.6` | `python3.7` | `dotnetcore1.0` | `dotnetcore2.1` | `go1.x` | `ruby2.5` | `provided`

Required: Yes

[Tags \(p. 355\)](#)

A list of [tags](#) to apply to the function.

Type: String to string map

Required: No

[Timeout \(p. 355\)](#)

The amount of time that Lambda allows a function to run before stopping it. The default is 3 seconds. The maximum allowed value is 900 seconds.

Type: Integer

Valid Range: Minimum value of 1.

Required: No

[TracingConfig \(p. 355\)](#)

Set `Mode` to `Active` to sample and trace a subset of incoming requests with AWS X-Ray.

Type: [TracingConfig \(p. 495\)](#) object

Required: No

[VpcConfig \(p. 355\)](#)

For network connectivity to AWS resources in a VPC, specify a list of security groups and subnets in the VPC. When you connect a function to a VPC, it can only access resources and the internet through that VPC. For more information, see [VPC Settings](#).

Type: [VpcConfig \(p. 497\)](#) object

Required: No

Response Syntax

```
HTTP/1.1 201
Content-type: application/json
```

```
{  
    "CodeSha256": "string",  
    "CodeSize": number,  
    "DeadLetterConfig": {  
        "TargetArn": "string"  
    },  
    "Description": "string",  
    "Environment": {  
        "Error": {  
            "ErrorCode": "string",  
            "Message": "string"  
        },  
        "Variables": {  
            "string" : "string"  
        }  
    },  
    "FunctionArn": "string",  
    "FunctionName": "string",  
    "Handler": "string",  
    "KMSKeyArn": "string",  
    "LastModified": "string",  
    "Layers": [  
        {  
            "Arn": "string",  
            "CodeSize": number  
        }  
    ],  
    "MasterArn": "string",  
    "MemorySize": number,  
    "RevisionId": "string",  
    "Role": "string",  
    "Runtime": "string",  
    "Timeout": number,  
    "TracingConfig": {  
        "Mode": "string"  
    },  
    "Version": "string",  
    "VpcConfig": {  
        "SecurityGroupIds": [ "string" ],  
        "SubnetIds": [ "string" ],  
        "VpcId": "string"  
    }  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

CodeSha256 (p. 358)

The SHA256 hash of the function's deployment package.

Type: String

CodeSize (p. 358)

The size of the function's deployment package, in bytes.

Type: Long

DeadLetterConfig (p. 358)

The function's dead letter queue.

Type: [DeadLetterConfig \(p. 477\)](#) object

Description (p. 358)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Environment (p. 358)

The function's environment variables.

Type: [EnvironmentResponse \(p. 480\)](#) object

FunctionArn (p. 358)

The function's Amazon Resource Name (ARN).

Type: String

Pattern: `arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

FunctionName (p. 358)

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: `(arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?\d{12}:?(function:)?([a-zA-Z0-9-_\.]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Handler (p. 358)

The function that Lambda calls to begin executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

KMSKeyArn (p. 358)

The KMS key that's used to encrypt the function's environment variables. This key is only returned if you've configured a customer-managed CMK.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*):[a-zA-Z0-9-_\.]+\.*)|()`

LastModified (p. 358)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Layers (p. 358)

The function's [layers](#).

Type: Array of [Layer \(p. 489\)](#) objects

[MasterArn \(p. 358\)](#)

For Lambda@Edge functions, the ARN of the master function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[MemorySize \(p. 358\)](#)

The memory that's allocated to the function.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 3008.

[RevisionId \(p. 358\)](#)

The latest updated revision of the function or alias.

Type: String

[Role \(p. 358\)](#)

The function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\-/]+`

[Runtime \(p. 358\)](#)

The runtime environment for the Lambda function.

Type: String

Valid Values: `nodejs8.10 | nodejs10.x | java8 | python2.7 | python3.6 | python3.7 | dotnetcore1.0 | dotnetcore2.1 | go1.x | ruby2.5 | provided`

[Timeout \(p. 358\)](#)

The amount of time that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

[TracingConfig \(p. 358\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 496\)](#) object

[Version \(p. 358\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

[VpcConfig \(p. 358\)](#)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 498\)](#) object

Errors

CodeStorageExceededException****

You have exceeded your maximum total code size per account. [Learn more](#)

HTTP Status Code: 400

InvalidParameterValueException****

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceConflictException****

The resource already exists.

HTTP Status Code: 409

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException****

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException****

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

DeleteAlias

Deletes a Lambda function [alias](#).

Request Syntax

```
DELETE /2015-03-31/functions/FunctionName/aliases/Name HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

[FunctionName](#) (p. 363)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:[a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

[Name](#) (p. 363)

The name of the alias.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (?![0-9]+\$)([a-zA-Z0-9-_]+)

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

[InvalidParameterValueException](#)

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

DeleteEventSourceMapping

Deletes an [event source mapping](#). You can get the identifier of a mapping from the output of [ListEventSourceMappings \(p. 410\)](#).

Request Syntax

```
DELETE /2015-03-31/event-source-mappings/UUID HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

UUID (p. 365)

The identifier of the event source mapping.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 202
Content-type: application/json

{
    "BatchSize": number,
    "EventSourceArn": "string",
    "FunctionArn": "string",
    "LastModified": number,
    "LastProcessingResult": "string",
    "State": "string",
    "StateTransitionReason": "string",
    "UUID": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 202 response.

The following data is returned in JSON format by the service.

BatchSize (p. 365)

The maximum number of items to retrieve in a single batch.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

EventSourceArn (p. 365)

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*)([a-zA-Z0-9\-])+([a-z]{2}(-gov)?-[a-z]+\-\d{1})?:(\d{12})?:(.*?)

FunctionArn (p. 365)

The ARN of the Lambda function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+\-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$\\$LATEST|[a-zA-Z0-9-_]+))?

LastModified (p. 365)

The date that the event source mapping was last updated, in Unix time seconds.

Type: Timestamp

LastProcessingResult (p. 365)

The result of the last AWS Lambda invocation of your Lambda function.

Type: String

State (p. 365)

The state of the event source mapping. It can be one of the following: Creating, Enabling, Enabled, Disabling, Disabled, Updating, or Deleting.

Type: String

StateTransitionReason (p. 365)

The cause of the last state change, either User initiated or Lambda initiated.

Type: String

UUID (p. 365)

The identifier of the event source mapping.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the CreateFunction or the UpdateFunctionConfiguration API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceInUseException

The operation conflicts with the resource's availability. For example, you attempted to update an EventSource Mapping in CREATING, or tried to delete a EventSource mapping currently in the UPDATING state.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

DeleteFunction

Deletes a Lambda function. To delete a specific function version, use the `Qualifier` parameter. Otherwise, all versions and aliases are deleted.

To delete Lambda event source mappings that invoke a function, use [DeleteEventSourceMapping \(p. 365\)](#). For AWS services and resources that invoke your function directly, delete the trigger in the service where you originally configured it.

Request Syntax

```
DELETE /2015-03-31/functions/FunctionName?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

[FunctionName \(p. 368\)](#)

The name of the Lambda function or version.

Name formats

- **Function name** - `my-function` (name-only), `my-function:1` (with version).
- **Function ARN** - `arn:aws:lambda:us-west-2:123456789012:function:my-function`.
- **Partial ARN** - `123456789012:function:my-function`.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[Qualifier \(p. 368\)](#)

Specify a version to delete. You can't delete a version that's referenced by an alias.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(|[a-zA-Z0-9$-_]+)`

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceConflictException

The resource already exists.

HTTP Status Code: 409

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

DeleteFunctionConcurrency

Removes a concurrent execution limit from a function.

Request Syntax

```
DELETE /2017-10-31/functions/FunctionName/concurrency HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 370)

The name of the Lambda function.

Name formats

- **Function name** - my-function.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

DeleteLayerVersion

Deletes a version of an [AWS Lambda layer](#). Deleted versions can no longer be viewed or added to functions. To avoid breaking functions, a copy of the version remains in Lambda until no functions refer to it.

Request Syntax

```
DELETE /2018-10-31/layers/LayerName/versions/VersionNumber HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

[LayerName \(p. 372\)](#)

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+)|[a-zA-Z0-9-_]+

[VersionNumber \(p. 372\)](#)

The version number.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

[ServiceException](#)

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

[TooManyRequestsException](#)

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

GetAccountSettings

Retrieves details about your account's [limits](#) and usage in an AWS Region.

Request Syntax

```
GET /2016-08-19/account-settings/ HTTP/1.1
```

URI Request Parameters

The request does not use any URI parameters.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "AccountLimit": {
        "CodeSizeUnzipped": number,
        "CodeSizeZipped": number,
        "ConcurrentExecutions": number,
        "TotalCodeSize": number,
        "UnreservedConcurrentExecutions": number
    },
    "AccountUsage": {
        "FunctionCount": number,
        "TotalCodeSize": number
    }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[AccountLimit \(p. 374\)](#)

Limits that are related to concurrency and code storage.

Type: [AccountLimit \(p. 471\)](#) object

[AccountUsage \(p. 374\)](#)

The number of functions and amount of storage in use.

Type: [AccountUsage \(p. 472\)](#) object

Errors

[ServiceException](#)

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

GetAlias

Returns details about a Lambda function [alias](#).

Request Syntax

```
GET /2015-03-31/functions/FunctionName/aliases/Name HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 376)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Name (p. 376)

The name of the alias.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (?![0-9]+\$)([a-zA-Z0-9-_]+)

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "AliasArn": "string",
    "Description": "string",
    "FunctionVersion": "string",
    "Name": "string",
    "RevisionId": "string",
    "RoutingConfig": {
        "AdditionalVersionWeights": {
            "string" : number
        }
    }
}
```

}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[AliasArn \(p. 376\)](#)

The Amazon Resource Name (ARN) of the alias.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?

[Description \(p. 376\)](#)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[FunctionVersion \(p. 376\)](#)

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\$LATEST|[0-9]+)

[Name \(p. 376\)](#)

The name of the alias.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (?![0-9]+\$)([a-zA-Z0-9-_]+)

[RevisionId \(p. 376\)](#)

A unique identifier that changes when you update the alias.

Type: String

[RoutingConfig \(p. 376\)](#)

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 475\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

GetEventSourceMapping

Returns details about an event source mapping. You can get the identifier of a mapping from the output of [ListEventSourceMappings \(p. 410\)](#).

Request Syntax

```
GET /2015-03-31/event-source-mappings/UUID HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

UUID (p. 379)

The identifier of the event source mapping.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "BatchSize": number,
    "EventSourceArn": "string",
    "FunctionArn": "string",
    "LastModified": number,
    "LastProcessingResult": "string",
    "State": "string",
    "StateTransitionReason": "string",
    "UUID": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

BatchSize (p. 379)

The maximum number of items to retrieve in a single batch.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

EventSourceArn (p. 379)

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+:(a-z{2}(-gov)?-[a-z]+\-\d{1})?:(\d{12})?:(.*)

[FunctionArn \(p. 379\)](#)

The ARN of the Lambda function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+\-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+:(\\$\\$LATEST|[a-zA-Z0-9-_]+))?

[LastModified \(p. 379\)](#)

The date that the event source mapping was last updated, in Unix time seconds.

Type: Timestamp

[LastProcessingResult \(p. 379\)](#)

The result of the last AWS Lambda invocation of your Lambda function.

Type: String

[State \(p. 379\)](#)

The state of the event source mapping. It can be one of the following: Creating, Enabling, Enabled, Disabling, Disabled, Updating, or Deleting.

Type: String

[StateTransitionReason \(p. 379\)](#)

The cause of the last state change, either User initiated or Lambda initiated.

Type: String

[UUID \(p. 379\)](#)

The identifier of the event source mapping.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

GetFunction

Returns information about the function or function version, with a link to download the deployment package that's valid for 10 minutes. If you specify a function version, only details that are specific to that version are returned.

Request Syntax

```
GET /2015-03-31/functions/FunctionName?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 382)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** - my-function (name-only), my-function:v1 (with alias).
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\\$\\$LATEST|[a-zA-Z0-9-_]+))?

Qualifier (p. 382)

Specify a version or alias to get details about a published version of the function.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$-_]+)

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "Code": {
        "Location": "string",
        "RepositoryType": "string"
    },
    "Concurrency": {
        "ReservedConcurrentExecutions": number
    }
}
```

```
},
"Configuration": {
    "CodeSha256": "string",
    "CodeSize": number,
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
    "Description": "string",
    "Environment": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "Variables": {
            "string" : "string"
        }
    },
    "FunctionArn": "string",
    "FunctionName": "string",
    "Handler": "string",
    "KMSKeyArn": "string",
    "LastModified": "string",
    "Layers": [
        {
            "Arn": "string",
            "CodeSize": number
        }
    ],
    "MasterArn": "string",
    "MemorySize": number,
    "RevisionId": "string",
    "Role": "string",
    "Runtime": "string",
    "Timeout": number,
    "TracingConfig": {
        "Mode": "string"
    },
    "Version": "string",
    "VpcConfig": {
        "SecurityGroupIds": [ "string" ],
        "SubnetIds": [ "string" ],
        "VpcId": "string"
    }
},
"Tags": {
    "string" : "string"
}
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Code (p. 382)

The deployment package of the function or version.

Type: [FunctionCodeLocation \(p. 484\)](#) object

Concurrency (p. 382)

The function's reserved concurrency.

Type: [Concurrency \(p. 476\)](#) object

[Configuration \(p. 382\)](#)

The configuration of the function or version.

Type: [FunctionConfiguration \(p. 485\)](#) object

[Tags \(p. 382\)](#)

The function's [tags](#).

Type: String to string map

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

GetFunctionConfiguration

Returns the version-specific settings of a Lambda function or version. The output includes only options that can vary between versions of a function. To modify these settings, use [UpdateFunctionConfiguration \(p. 463\)](#).

To get all of a function's details, including function-level settings, use [GetFunction \(p. 382\)](#).

Request Syntax

```
GET /2015-03-31/functions/FunctionName/configuration?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

[FunctionName \(p. 385\)](#)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** - my-function (name-only), my-function:v1 (with alias).
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}):?(function:)?([a-zA-Z0-9-_\.]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

[Qualifier \(p. 385\)](#)

Specify a version or alias to get details about a published version of the function.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$-_]+)

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "CodeSha256": "string",
    "CodeSize": number,
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
}
```

```
"Description": "string",
"Environment": {
    "Error": {
        "ErrorCode": "string",
        "Message": "string"
    },
    "Variables": {
        "string" : "string"
    }
},
"FunctionArn": "string",
"FunctionName": "string",
"Handler": "string",
"KMSKeyArn": "string",
"LastModified": "string",
"Layers": [
    {
        "Arn": "string",
        "CodeSize": number
    }
],
"MasterArn": "string",
"MemorySize": number,
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"Timeout": number,
"TracingConfig": {
    "Mode": "string"
},
"Version": "string",
"VpcConfig": {
    "SecurityGroupIds": [ "string" ],
    "SubnetIds": [ "string" ],
    "VpcId": "string"
}
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[CodeSha256 \(p. 385\)](#)

The SHA256 hash of the function's deployment package.

Type: String

[CodeSize \(p. 385\)](#)

The size of the function's deployment package, in bytes.

Type: Long

[DeadLetterConfig \(p. 385\)](#)

The function's dead letter queue.

Type: [DeadLetterConfig \(p. 477\)](#) object

[Description \(p. 385\)](#)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[Environment \(p. 385\)](#)

The function's environment variables.

Type: [EnvironmentResponse \(p. 480\)](#) object

[FunctionArn \(p. 385\)](#)

The function's Amazon Resource Name (ARN).

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[FunctionName \(p. 385\)](#)

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?\d{12}:?function:[a-zA-Z0-9-_\.]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[Handler \(p. 385\)](#)

The function that Lambda calls to begin executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

[KMSKeyArn \(p. 385\)](#)

The KMS key that's used to encrypt the function's environment variables. This key is only returned if you've configured a customer-managed CMK.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-_\.]+\.*|()`

[LastModified \(p. 385\)](#)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[Layers \(p. 385\)](#)

The function's [layers](#).

Type: Array of [Layer \(p. 489\)](#) objects

[MasterArn \(p. 385\)](#)

For Lambda@Edge functions, the ARN of the master function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[MemorySize \(p. 385\)](#)

The memory that's allocated to the function.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 3008.

[RevisionId \(p. 385\)](#)

The latest updated revision of the function or alias.

Type: String

[Role \(p. 385\)](#)

The function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@/-/_]+`

[Runtime \(p. 385\)](#)

The runtime environment for the Lambda function.

Type: String

Valid Values: nodejs8.10 | nodejs10.x | java8 | python2.7 | python3.6 | python3.7 | dotnetcore1.0 | dotnetcore2.1 | go1.x | ruby2.5 | provided

[Timeout \(p. 385\)](#)

The amount of time that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

[TracingConfig \(p. 385\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 496\)](#) object

[Version \(p. 385\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

[VpcConfig \(p. 385\)](#)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 498\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

GetLayerVersion

Returns information about a version of an AWS Lambda layer, with a link to download the layer archive that's valid for 10 minutes.

Request Syntax

```
GET /2018-10-31/layers/LayerName/versions/VersionNumber HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

LayerName (p. 390)

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+\:\d{12}:layer:[a-zA-Z0-9-_+])|[a-zA-Z0-9-_+]

VersionNumber (p. 390)

The version number.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "CompatibleRuntimes": [ "string" ],
  "Content": {
    "CodeSha256": "string",
    "CodeSize": number,
    "Location": "string"
  },
  "CreatedDate": "string",
  "Description": "string",
  "LayerArn": "string",
  "LayerVersionArn": "string",
  "LicenseInfo": "string",
  "Version": number
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[CompatibleRuntimes \(p. 390\)](#)

The layer's compatible runtimes.

Type: Array of strings

Array Members: Maximum number of 5 items.

Valid Values: nodejs8.10 | nodejs10.x | java8 | python2.7 | python3.6 | python3.7 | dotnetcore1.0 | dotnetcore2.1 | go1.x | ruby2.5 | provided

[Content \(p. 390\)](#)

Details about the layer version.

Type: [LayerVersionContentOutput \(p. 492\)](#) object

[CreatedDate \(p. 390\)](#)

The date that the layer version was created, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[Description \(p. 390\)](#)

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[LayerArn \(p. 390\)](#)

The ARN of the layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+

[LayerVersionArn \(p. 390\)](#)

The ARN of the layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+

[LicenseInfo \(p. 390\)](#)

The layer's software license.

Type: String

Length Constraints: Maximum length of 512.

[Version \(p. 390\)](#)

The version number.

Type: Long

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

GetLayerVersionByArn

Returns information about a version of an AWS Lambda layer, with a link to download the layer archive that's valid for 10 minutes.

Request Syntax

```
GET /2018-10-31/layers?find=LayerVersion&Arn=Arn HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

Arn (p. 393)

The ARN of the layer version.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "CompatibleRuntimes": [ "string" ],
    "Content": {
        "CodeSha256": "string",
        "CodeSize": number,
        "Location": "string"
    },
    "CreatedDate": "string",
    "Description": "string",
    "LayerArn": "string",
    "LayerVersionArn": "string",
    "LicenseInfo": "string",
    "Version": number
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

CompatibleRuntimes (p. 393)

The layer's compatible runtimes.

Type: Array of strings

Array Members: Maximum number of 5 items.

Valid Values: nodejs8.10 | nodejs10.x | java8 | python2.7 | python3.6 | python3.7 | dotnetcore1.0 | dotnetcore2.1 | go1.x | ruby2.5 | provided

[Content \(p. 393\)](#)

Details about the layer version.

Type: [LayerVersionContentOutput \(p. 492\)](#) object

[CreatedDate \(p. 393\)](#)

The date that the layer version was created, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[Description \(p. 393\)](#)

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[LayerArn \(p. 393\)](#)

The ARN of the layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+

[LayerVersionArn \(p. 393\)](#)

The ARN of the layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+

[LicenseInfo \(p. 393\)](#)

The layer's software license.

Type: String

Length Constraints: Maximum length of 512.

[Version \(p. 393\)](#)

The version number.

Type: Long

Errors

InvalidArgumentException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

GetLayerVersionPolicy

Returns the permission policy for a version of an AWS Lambda layer. For more information, see [AddLayerVersionPermission \(p. 339\)](#).

Request Syntax

```
GET /2018-10-31/layers/LayerName/versions/VersionNumber/policy HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

LayerName (p. 396)

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_+])|[a-zA-Z0-9-_+]

VersionNumber (p. 396)

The version number.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "Policy": "string",
    "RevisionId": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Policy (p. 396)

The policy document.

Type: String

RevisionId (p. 396)

A unique identifier for the current revision of the policy.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

GetPolicy

Returns the resource-based IAM policy for a function, version, or alias.

Request Syntax

```
GET /2015-03-31/functions/FunctionName/policy?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 398)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** - my-function (name-only), my-function:v1 (with alias).
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Qualifier (p. 398)

Specify a version or alias to get the policy for that resource.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (| [a-zA-Z0-9\$-_]+)

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "Policy": "string",
    "RevisionId": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Policy \(p. 398\)](#)

The resource-based policy.

Type: String

[RevisionId \(p. 398\)](#)

A unique identifier for the current revision of the policy.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

Invoke

Invokes a Lambda function. You can invoke a function synchronously (and wait for the response), or asynchronously. To invoke a function asynchronously, set `InvocationType` to `Event`.

For synchronous invocation, details about the function response, including errors, are included in the response body and headers. For either invocation type, you can find more information in the [execution log](#) and [trace](#). To record function errors for asynchronous invocations, configure your function with a [dead letter queue](#).

When an error occurs, your function may be invoked multiple times. Retry behavior varies by error type, client, event source, and invocation type. For example, if you invoke a function asynchronously and it returns an error, Lambda executes the function up to two more times. For more information, see [Retry Behavior](#).

The status code in the API response doesn't reflect function errors. Error codes are reserved for errors that prevent your function from executing, such as permissions errors, [limit errors](#), or issues with your function's code and configuration. For example, Lambda returns `TooManyRequestsException` if executing the function would cause you to exceed a concurrency limit at either the account level (`ConcurrentInvocationLimitExceeded`) or function level (`ReservedFunctionConcurrentInvocationLimitExceeded`).

For functions with a long timeout, your client might be disconnected during synchronous invocation while it waits for a response. Configure your HTTP client, SDK, firewall, proxy, or operating system to allow for long connections with timeout or keep-alive settings.

This operation requires permission for the `lambda:InvokeFunction` action.

Request Syntax

```
POST /2015-03-31/functions/FunctionName/invocations?Qualifier=Qualifier HTTP/1.1
X-Amz-Invocation-Type: InvocationType
X-Amz-Log-Type: LogType
X-Amz-Client-Context: ClientContext

Payload
```

URI Request Parameters

The request requires the following URI parameters.

[ClientContext \(p. 400\)](#)

Up to 3583 bytes of base64-encoded data about the invoking client to pass to the function in the `context` object.

[FunctionName \(p. 400\)](#)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** - `my-function` (name-only), `my-function:v1` (with alias).
- **Function ARN** - `arn:aws:lambda:us-west-2:123456789012:function:my-function`.
- **Partial ARN** - `123456789012:function:my-function`.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (`arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

InvocationType (p. 400)

Choose from the following options.

- `RequestResponse` (default) - Invoke the function synchronously. Keep the connection open until the function returns a response or times out. The API response includes the function response and additional data.
- `Event` - Invoke the function asynchronously. Send events that fail multiple times to the function's dead-letter queue (if it's configured). The API response only includes a status code.
- `DryRun` - Validate parameter values and verify that the user or role has permission to invoke the function.

Valid Values: `Event` | `RequestResponse` | `DryRun`

LogType (p. 400)

Set to `Tail` to include the execution log in the response.

Valid Values: `None` | `Tail`

Qualifier (p. 400)

Specify a version or alias to invoke a published version of the function.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (`|[a-zA-Z0-9$-_]+`)

Request Body

The request accepts the following binary data.

Payload (p. 400)

The JSON that you want to provide to your Lambda function as input.

Response Syntax

```
HTTP/1.1 StatusCode
X-Amz-Function-Error: FunctionError
X-Amz-Log-Result: LogResult
X-Amz-Executed-Version: ExecutedVersion

Payload
```

Response Elements

If the action is successful, the service sends back the following HTTP response.

StatusCode (p. 401)

The HTTP status code is in the 200 range for a successful request. For the `RequestResponse` invocation type, this status code is 200. For the `Event` invocation type, this status code is 202. For the `DryRun` invocation type, the status code is 204.

The response returns the following HTTP headers.

[ExecutedVersion \(p. 401\)](#)

The version of the function that executed. When you invoke a function with an alias, this indicates which version the alias resolved to.

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (`\$LATEST` | [0-9]+)

[FunctionError \(p. 401\)](#)

If present, indicates that an error occurred during function execution. Details about the error are included in the response payload.

- **Handled** - The runtime caught an error thrown by the function and formatted it into a JSON document.
- **Unhandled** - The runtime didn't handle the error. For example, the function ran out of memory or timed out.

[LogResult \(p. 401\)](#)

The last 4 KB of the execution log, which is base64 encoded.

The response returns the following as the HTTP body.

[Payload \(p. 401\)](#)

The response from the function, or an error object.

Errors

EC2AccessDeniedException

Need additional permissions to configure VPC settings.

HTTP Status Code: 502

EC2ThrottledException

AWS Lambda was throttled by Amazon EC2 during Lambda function initialization using the execution role provided for the Lambda function.

HTTP Status Code: 502

EC2UnexpectedException

AWS Lambda received an unexpected EC2 client exception while setting up for the Lambda function.

HTTP Status Code: 502

ENILimitReachedException

AWS Lambda was not able to create an Elastic Network Interface (ENI) in the VPC, specified as part of Lambda function configuration, because the limit for network interfaces has been reached.

HTTP Status Code: 502

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

InvalidRequestContentException

The request body could not be parsed as JSON.

HTTP Status Code: 400

InvalidRuntimeException

The runtime or runtime version specified is not supported.

HTTP Status Code: 502

InvalidSecurityGroupIDException

The Security Group ID provided in the Lambda function VPC configuration is invalid.

HTTP Status Code: 502

InvalidSubnetIDException

The Subnet ID provided in the Lambda function VPC configuration is invalid.

HTTP Status Code: 502

InvalidZipFileException

AWS Lambda could not unzip the deployment package.

HTTP Status Code: 502

KMSAccessDeniedException

Lambda was unable to decrypt the environment variables because KMS access was denied. Check the Lambda function's KMS permissions.

HTTP Status Code: 502

KMSDisabledException

Lambda was unable to decrypt the environment variables because the KMS key used is disabled. Check the Lambda function's KMS key settings.

HTTP Status Code: 502

KMSInvalidStateException

Lambda was unable to decrypt the environment variables because the KMS key used is in an invalid state for Decrypt. Check the function's KMS key settings.

HTTP Status Code: 502

KMSNotFoundException

Lambda was unable to decrypt the environment variables because the KMS key was not found. Check the function's KMS key settings.

HTTP Status Code: 502

RequestTooLargeException

The request payload exceeded the [Invoke](#) request body JSON input limit. For more information, see [Limits](#).

HTTP Status Code: 413

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

SubnetIPAddressLimitReachedException

AWS Lambda was not able to set up VPC access for the Lambda function because one or more configured subnets has no available IP addresses.

HTTP Status Code: 502

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

UnsupportedMediaTypeException

The content type of the `Invoke` request body is not JSON.

HTTP Status Code: 415

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

InvokeAsync

This action has been deprecated.

Important

For asynchronous function invocation, use [Invoke \(p. 400\)](#).

Invokes a function asynchronously.

Request Syntax

```
POST /2014-11-13/functions/FunctionName/invoke-async/ HTTP/1.1
```

```
InvokeArgs
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 405)

The name of the Lambda function.

Name formats

- **Function name** - my-function.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Request Body

The request accepts the following binary data.

InvokeArgs (p. 405)

The JSON that you want to provide to your Lambda function as input.

Response Syntax

```
HTTP/1.1 Status
```

Response Elements

If the action is successful, the service sends back the following HTTP response.

[Status \(p. 405\)](#)

The status code.

Errors

InvalidRequestContentException

The request body could not be parsed as JSON.

HTTP Status Code: 400

InvalidRuntimeException

The runtime or runtime version specified is not supported.

HTTP Status Code: 502

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

ListAliases

Returns a list of [aliases](#) for a Lambda function.

Request Syntax

```
GET /2015-03-31/functions/FunctionName/aliases?  
FunctionVersion=FunctionVersion&Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

[FunctionName \(p. 407\)](#)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

[FunctionVersion \(p. 407\)](#)

Specify a function version to only list aliases that invoke that version.

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST|[0-9]+)

[Marker \(p. 407\)](#)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MaxItems \(p. 407\)](#)

Limit the number of aliases returned.

Valid Range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200  
Content-type: application/json
```

```
{  
    "Aliases": [  
        {  
            "AliasArn": "string",  
            "Description": "string",  
            "FunctionVersion": "string",  
            "Name": "string",  
            "RevisionId": "string",  
            "RoutingConfig": {  
                "AdditionalVersionWeights": {  
                    "string" : number  
                }  
            }  
        }  
    ],  
    "NextMarker": "string"  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Aliases \(p. 407\)](#)

A list of aliases.

Type: Array of [AliasConfiguration \(p. 473\)](#) objects

[NextMarker \(p. 407\)](#)

The pagination token that's included if more results are available.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

ListEventSourceMappings

Lists event source mappings. Specify an `EventSourceArn` to only show event source mappings for a single event source.

Request Syntax

```
GET /2015-03-31/event-source-mappings/?  
EventSourceArn=EventSourceArn&FunctionName=FunctionName&Marker=Marker&MaxItems=MaxItems  
HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

`EventSourceArn` (p. 410)

The Amazon Resource Name (ARN) of the event source.

- **Amazon Kinesis** - The ARN of the data stream or a stream consumer.
- **Amazon DynamoDB Streams** - The ARN of the stream.
- **Amazon Simple Queue Service** - The ARN of the queue.

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9-]+):([a-z]{2}(-gov)?-[a-z]+\d{1}):(\d{12}):(.*)`

`FunctionName` (p. 410)

The name of the Lambda function.

Name formats

- **Function name** - `MyFunction`.
- **Function ARN** - `arn:aws:lambda:us-west-2:123456789012:function:MyFunction`.
- **Version or Alias ARN** - `arn:aws:lambda:us-west-2:123456789012:function:MyFunction:PROD`.
- **Partial ARN** - `123456789012:function:MyFunction`.

The length constraint applies only to the full ARN. If you specify only the function name, it's limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:)(\d{12}:)(function:[a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

`Marker` (p. 410)

A pagination token returned by a previous call.

`MaxItems` (p. 410)

The maximum number of event source mappings to return.

Valid Range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "EventSourceMappings": [
    {
      "BatchSize": number,
      "EventSourceArn": "string",
      "FunctionArn": "string",
      "LastModified": number,
      "LastProcessingResult": "string",
      "State": "string",
      "StateTransitionReason": "string",
      "UUID": "string"
    }
  ],
  "NextMarker": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

EventSourceMappings (p. 411)

A list of event source mappings.

Type: Array of [EventSourceMappingConfiguration \(p. 481\)](#) objects

NextMarker (p. 411)

A pagination token that's returned when the response doesn't contain all event source mappings.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

ListFunctions

Returns a list of Lambda functions, with the version-specific configuration of each.

Set `FunctionVersion` to `ALL` to include all published versions of each function in addition to the unpublished version. To get more information about a function or version, use [GetFunction \(p. 382\)](#).

Request Syntax

```
GET /2015-03-31/functions/?  
FunctionVersion=FunctionVersion&Marker=Marker&MasterRegion=MasterRegion&MaxItems=MaxItems  
HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

[FunctionVersion \(p. 413\)](#)

Set to `ALL` to include entries for all published versions of each function.

Valid Values: `ALL`

[Marker \(p. 413\)](#)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MasterRegion \(p. 413\)](#)

For Lambda@Edge functions, the AWS Region of the master function. For example, `us-east-2` or `ALL`. If specified, you must set `FunctionVersion` to `ALL`.

Pattern: `ALL | [a-z]{2}(-gov)?-[a-z]+-\d{1}`

[MaxItems \(p. 413\)](#)

Specify a value between 1 and 50 to limit the number of functions in the response.

Valid Range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200  
Content-type: application/json  
  
{  
  "Functions": [  
    {  
      "CodeSha256": "string",  
      "CodeSize": number,  
      "DeadLetterConfig": {  
        "TargetArn": "string"  
      },  
      "Description": "string",  
      "Environment": {  
        "Error": {  
          "Code": "string",  
          "Message": "string"  
        }  
      },  
      "FunctionArn": "string",  
      "Handler": "string",  
      "LastModified": "string",  
      "MemorySize": number,  
      "Runtime": "string",  
      "State": "string",  
      "Timeout": number  
    }  
  ]  
}  
}
```

```
        "ErrorCode": "string",
        "Message": "string"
    },
    "Variables": {
        "string" : "string"
    }
},
"FunctionArn": "string",
"FunctionName": "string",
"Handler": "string",
"KMSKeyArn": "string",
"LastModified": "string",
"Layers": [
    {
        "Arn": "string",
        "CodeSize": number
    }
],
"MasterArn": "string",
"MemorySize": number,
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"Timeout": number,
"TracingConfig": {
    "Mode": "string"
},
"Version": "string",
"VpcConfig": {
    "SecurityGroupIds": [ "string" ],
    "SubnetIds": [ "string" ],
    "VpcId": "string"
}
],
"NextMarker": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Functions \(p. 413\)](#)

A list of Lambda functions.

Type: Array of [FunctionConfiguration \(p. 485\)](#) objects

[NextMarker \(p. 413\)](#)

The pagination token that's included if more results are available.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

ListLayers

Lists [AWS Lambda layers](#) and shows information about the latest version of each. Specify a [runtime identifier](#) to list only layers that indicate that they're compatible with that runtime.

Request Syntax

```
GET /2018-10-31/layers?CompatibleRuntime=CompatibleRuntime&Marker=Marker&MaxItems=MaxItems
HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

CompatibleRuntime ([p. 416](#))

A runtime identifier. For example, go1.x.

Valid Values: nodejs8.10 | nodejs10.x | java8 | python2.7 | python3.6 | python3.7 | dotnetcore1.0 | dotnetcore2.1 | go1.x | ruby2.5 | provided

Marker ([p. 416](#))

A pagination token returned by a previous call.

MaxItems ([p. 416](#))

The maximum number of layers to return.

Valid Range: Minimum value of 1. Maximum value of 50.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "Layers": [
    {
      "LatestMatchingVersion": {
        "CompatibleRuntimes": [ "string" ],
        "CreatedDate": "string",
        "Description": "string",
        "LayerVersionArn": "string",
        "LicenseInfo": "string",
        "Version": number
      },
      "LayerArn": "string",
      "LayerName": "string"
    }
  ],
  "NextMarker": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Layers \(p. 416\)](#)

A list of function layers.

Type: Array of [LayersListItem \(p. 490\)](#) objects

[NextMarker \(p. 416\)](#)

A pagination token returned when the response doesn't contain all layers.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

ListLayerVersions

Lists the versions of an [AWS Lambda layer](#). Versions that have been deleted aren't listed. Specify a [runtime identifier](#) to list only versions that indicate that they're compatible with that runtime.

Request Syntax

```
GET /2018-10-31/layers/LayerName/versions?  
CompatibleRuntime=CompatibleRuntime&Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

CompatibleRuntime ([p. 418](#))

A runtime identifier. For example, go1.x.

Valid Values: nodejs8.10 | nodejs10.x | java8 | python2.7 | python3.6 | python3.7 | dotnetcore1.0 | dotnetcore2.1 | go1.x | ruby2.5 | provided

LayerName ([p. 418](#))

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_+])|[a-zA-Z0-9-_+]

Marker ([p. 418](#))

A pagination token returned by a previous call.

MaxItems ([p. 418](#))

The maximum number of versions to return.

Valid Range: Minimum value of 1. Maximum value of 50.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200  
Content-type: application/json  
  
{  
    "LayerVersions": [  
        {  
            "CompatibleRuntimes": [ "string" ],  
            "CreatedDate": "string",  
            "Description": "string",  
            "LayerVersionArn": "string",  
            "LicenseInfo": "string",  
            "Version": number  
        }  
    ]  
}
```

```
    ],
    "NextMarker": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[LayerVersions \(p. 418\)](#)

A list of versions.

Type: Array of [LayerVersionsListItem \(p. 493\)](#) objects

[NextMarker \(p. 418\)](#)

A pagination token returned when the response doesn't contain all versions.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)

- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

ListTags

Returns a function's [tags](#). You can also view tags with [GetFunction \(p. 382\)](#).

Request Syntax

```
GET /2017-03-31/tags/ARN HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

[Resource \(p. 421\)](#)

The function's Amazon Resource Name (ARN).

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "Tags": {
    "string" : "string"
  }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Tags \(p. 421\)](#)

The function's tags.

Type: String to string map

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

ListVersionsByFunction

Returns a list of [versions](#), with the version-specific configuration of each.

Request Syntax

```
GET /2015-03-31/functions/FunctionName/versions?Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

[FunctionName](#) (p. 423)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

[Marker](#) (p. 423)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MaxItems](#) (p. 423)

Limit the number of versions that are returned.

Valid Range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "NextMarker": "string",
  "Versions": [
    {
      "Version": "string",
      "LastModified": "2015-03-31T22:21:15.000Z",
      "Runtime": "nodejs4.3",
      "CodeSize": 123,
      "MemorySize": 128,
      "Timeout": 3,
      "TracingConfig": {
        "SamplingPercentage": 100
      }
    }
  ]
}
```

```
"CodeSha256": "string",
"CodeSize": number,
"DeadLetterConfig": {
    "TargetArn": "string"
},
"Description": "string",
"Environment": {
    "Error": {
        "ErrorCode": "string",
        "Message": "string"
    },
    "Variables": {
        "string" : "string"
    }
},
"FunctionArn": "string",
"FunctionName": "string",
"Handler": "string",
"KMSKeyArn": "string",
"LastModified": "string",
"Layers": [
    {
        "Arn": "string",
        "CodeSize": number
    }
],
"MasterArn": "string",
"MemorySize": number,
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"Timeout": number,
"TracingConfig": {
    "Mode": "string"
},
"Version": "string",
"VpcConfig": {
    "SecurityGroupIds": [ "string" ],
    "SubnetIds": [ "string" ],
    "VpcId": "string"
}
}
]
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[NextMarker \(p. 423\)](#)

The pagination token that's included if more results are available.

Type: String

[Versions \(p. 423\)](#)

A list of Lambda function versions.

Type: Array of [FunctionConfiguration \(p. 485\)](#) objects

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

PublishLayerVersion

Creates an [AWS Lambda layer](#) from a ZIP archive. Each time you call `PublishLayerVersion` with the same version name, a new version is created.

Add layers to your function with [CreateFunction \(p. 355\)](#) or [UpdateFunctionConfiguration \(p. 463\)](#).

Request Syntax

```
POST /2018-10-31/layers/LayerName/versions HTTP/1.1
Content-type: application/json

{
  "CompatibleRuntimes": [ "string" ],
  "Content": {
    "S3Bucket": "string",
    "S3Key": "string",
    "S3ObjectVersion": "string",
    "ZipFile": blob
  },
  "Description": "string",
  "LicenseInfo": "string"
}
```

URI Request Parameters

The request requires the following URI parameters.

LayerName (p. 426)

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (`arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+| [a-zA-Z0-9-_]+`)

Request Body

The request accepts the following data in JSON format.

CompatibleRuntimes (p. 426)

A list of compatible [function runtimes](#). Used for filtering with [ListLayers \(p. 416\)](#) and [ListLayerVersions \(p. 418\)](#).

Type: Array of strings

Array Members: Maximum number of 5 items.

Valid Values: nodejs8.10 | nodejs10.x | java8 | python2.7 | python3.6 | python3.7 | dotnetcore1.0 | dotnetcore2.1 | go1.x | ruby2.5 | provided

Required: No

Content (p. 426)

The function layer archive.

Type: [LayerVersionContentInput \(p. 491\)](#) object

Required: Yes

[Description \(p. 426\)](#)

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

[LicenseInfo \(p. 426\)](#)

The layer's software license. It can be any of the following:

- An [SPDX license identifier](#). For example, `MIT`.
- The URL of a license hosted on the internet. For example, <https://opensource.org/licenses/MIT>.
- The full text of the license.

Type: String

Length Constraints: Maximum length of 512.

Required: No

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
    "CompatibleRuntimes": [ "string" ],
    "Content": {
        "CodeSha256": "string",
        "CodeSize": number,
        "Location": "string"
    },
    "CreatedDate": "string",
    "Description": "string",
    "LayerArn": "string",
    "LayerVersionArn": "string",
    "LicenseInfo": "string",
    "Version": number
}
```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

[CompatibleRuntimes \(p. 427\)](#)

The layer's compatible runtimes.

Type: Array of strings

Array Members: Maximum number of 5 items.

Valid Values: nodejs8.10 | nodejs10.x | java8 | python2.7 | python3.6 | python3.7 | dotnetcore1.0 | dotnetcore2.1 | go1.x | ruby2.5 | provided

Content (p. 427)

Details about the layer version.

Type: [LayerVersionContentOutput \(p. 492\)](#) object

CreatedDate (p. 427)

The date that the layer version was created, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Description (p. 427)

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

LayerArn (p. 427)

The ARN of the layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+

LayerVersionArn (p. 427)

The ARN of the layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+

LicenseInfo (p. 427)

The layer's software license.

Type: String

Length Constraints: Maximum length of 512.

Version (p. 427)

The version number.

Type: Long

Errors

CodeStorageExceededException****

You have exceeded your maximum total code size per account. [Learn more](#)

HTTP Status Code: 400

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

PublishVersion

Creates a [version](#) from the current code and configuration of a function. Use versions to create a snapshot of your function code and configuration that doesn't change.

AWS Lambda doesn't publish a version if the function's configuration and code haven't changed since the last version. Use [UpdateFunctionCode \(p. 456\)](#) or [UpdateFunctionConfiguration \(p. 463\)](#) to update the function before publishing a version.

Clients can invoke versions directly or with an alias. To create an alias, use [CreateAlias \(p. 347\)](#).

Request Syntax

```
POST /2015-03-31/functions/FunctionName/versions HTTP/1.1
Content-type: application/json

{
  "CodeSha256": "string",
  "Description": "string",
  "RevisionId": "string"
}
```

URI Request Parameters

The request requires the following URI parameters.

[FunctionName \(p. 430\)](#)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Request Body

The request accepts the following data in JSON format.

[CodeSha256 \(p. 430\)](#)

Only publish a version if the hash value matches the value that's specified. Use this option to avoid publishing a version if the function code has changed since you last updated it. You can get the hash for the version that you uploaded from the output of [UpdateFunctionCode \(p. 456\)](#).

Type: String

Required: No

Description (p. 430)

A description for the version to override the description in the function configuration.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

RevisionId (p. 430)

Only update the function if the revision ID matches the ID that's specified. Use this option to avoid publishing a version if the function configuration has changed since you last updated it.

Type: String

Required: No

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
    "CodeSha256": "string",
    "CodeSize": number,
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
    "Description": "string",
    "Environment": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "Variables": {
            "string" : "string"
        }
    },
    "FunctionArn": "string",
    "FunctionName": "string",
    "Handler": "string",
    "KMSKeyArn": "string",
    "LastModified": "string",
    "Layers": [
        {
            "Arn": "string",
            "CodeSize": number
        }
    ],
    "MasterArn": "string",
    "MemorySize": number,
    "RevisionId": "string",
    "Role": "string",
    "Runtime": "string",
    "Timeout": number,
    "TracingConfig": {
        "Mode": "string"
    },
    "Version": "string",
    "VpcConfig": {
        "SecurityGroupIds": [ "string" ],
        "SubnetIds": [ "string" ]
    }
}
```

```
    "SubnetIds": [ "string" ],
    "VpcId": "string"
}
}
```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

[CodeSha256 \(p. 431\)](#)

The SHA256 hash of the function's deployment package.

Type: String

[CodeSize \(p. 431\)](#)

The size of the function's deployment package, in bytes.

Type: Long

[DeadLetterConfig \(p. 431\)](#)

The function's dead letter queue.

Type: [DeadLetterConfig \(p. 477\)](#) object

[Description \(p. 431\)](#)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[Environment \(p. 431\)](#)

The function's environment variables.

Type: [EnvironmentResponse \(p. 480\)](#) object

[FunctionArn \(p. 431\)](#)

The function's Amazon Resource Name (ARN).

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:(\\$\\$LATEST|[a-zA-Z0-9-_]+))?

[FunctionName \(p. 431\)](#)

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\\$\\$LATEST|[a-zA-Z0-9-_]+))?

[Handler \(p. 431\)](#)

The function that Lambda calls to begin executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: [^\s]+

[KMSKeyArn \(p. 431\)](#)

The KMS key that's used to encrypt the function's environment variables. This key is only returned if you've configured a customer-managed CMK.

Type: String

Pattern: (arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-.]+:[.*])|()

[LastModified \(p. 431\)](#)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[Layers \(p. 431\)](#)

The function's [layers](#).

Type: Array of [Layer \(p. 489\)](#) objects

[MasterArn \(p. 431\)](#)

For Lambda@Edge functions, the ARN of the master function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$LATEST|[a-zA-Z0-9-_]+))?

[MemorySize \(p. 431\)](#)

The memory that's allocated to the function.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 3008.

[RevisionId \(p. 431\)](#)

The latest updated revision of the function or alias.

Type: String

[Role \(p. 431\)](#)

The function's execution role.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@/-/_/+]

[Runtime \(p. 431\)](#)

The runtime environment for the Lambda function.

Type: String

Valid Values: nodejs8.10 | nodejs10.x | java8 | python2.7 | python3.6 | python3.7 | dotnetcore1.0 | dotnetcore2.1 | go1.x | ruby2.5 | provided

[Timeout \(p. 431\)](#)

The amount of time that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

[TracingConfig \(p. 431\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 496\)](#) object

[Version \(p. 431\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (`\$LATEST` | [0-9]+)

[VpcConfig \(p. 431\)](#)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 498\)](#) object

Errors

CodeStorageExceededException

You have exceeded your maximum total code size per account. [Learn more](#)

HTTP Status Code: 400

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

PreconditionFailedException

The `RevisionId` provided does not match the latest `RevisionId` for the Lambda function or alias. Call the `GetFunction` or the `GetAlias` API to retrieve the latest `RevisionId` for your resource.

HTTP Status Code: 412

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

PutFunctionConcurrency

Sets the maximum number of simultaneous executions for a function, and reserves capacity for that concurrency level.

Concurrency settings apply to the function as a whole, including all published versions and the unpublished version. Reserving concurrency both ensures that your function has capacity to process the specified number of events simultaneously, and prevents it from scaling beyond that level. Use [GetFunction \(p. 382\)](#) to see the current setting for a function.

Use [GetAccountSettings \(p. 374\)](#) to see your regional concurrency limit. You can reserve concurrency for as many functions as you like, as long as you leave at least 100 simultaneous executions unreserved for functions that aren't configured with a per-function limit. For more information, see [Managing Concurrency](#).

Request Syntax

```
PUT /2017-10-31/functions/FunctionName/concurrency HTTP/1.1
Content-type: application/json

{
    "ReservedConcurrentExecutions": number
}
```

URI Request Parameters

The request requires the following URI parameters.

[FunctionName \(p. 436\)](#)

The name of the Lambda function.

Name formats

- **Function name** - my-function.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?:([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Request Body

The request accepts the following data in JSON format.

[ReservedConcurrentExecutions \(p. 436\)](#)

The number of simultaneous executions to reserve for the function.

Type: Integer

Valid Range: Minimum value of 0.

Required: Yes

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "ReservedConcurrentExecutions": number
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[ReservedConcurrentExecutions \(p. 437\)](#)

The number of concurrent executions that are reserved for this function. For more information, see [Managing Concurrency](#).

Type: Integer

Valid Range: Minimum value of 0.

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

RemoveLayerVersionPermission

Removes a statement from the permissions policy for a version of an AWS Lambda layer. For more information, see [AddLayerVersionPermission \(p. 339\)](#).

Request Syntax

```
DELETE /2018-10-31/layers/LayerName/versions/VersionNumber/policy/StatementId?  
RevisionId=RevisionId HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

[LayerName \(p. 439\)](#)

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+)|[a-zA-Z0-9-_]+

[RevisionId \(p. 439\)](#)

Only update the policy if the revision ID matches the ID specified. Use this option to avoid modifying a policy that has changed since you last read it.

[StatementId \(p. 439\)](#)

The identifier that was specified when the statement was added.

Length Constraints: Minimum length of 1. Maximum length of 100.

Pattern: ([a-zA-Z0-9-_]+)

[VersionNumber \(p. 439\)](#)

The version number.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

[InvalidParameterValueException](#)

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

PreconditionFailedException

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

RemovePermission

Revokes function-use permission from an AWS service or another account. You can get the ID of the statement from the output of [GetPolicy \(p. 398\)](#).

Request Syntax

```
DELETE /2015-03-31/functions/FunctionName/policy/StatementId?  
Qualifier=Qualifier&RevisionId=RevisionId HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

[FunctionName \(p. 441\)](#)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** - my-function (name-only), my-function:v1 (with alias).
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

[Qualifier \(p. 441\)](#)

Specify a version or alias to remove permissions from a published version of the function.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$-_]+)

[RevisionId \(p. 441\)](#)

Only update the policy if the revision ID matches the ID that's specified. Use this option to avoid modifying a policy that has changed since you last read it.

[StatementId \(p. 441\)](#)

Statement ID of the permission to remove.

Length Constraints: Minimum length of 1. Maximum length of 100.

Pattern: ([a-zA-Z0-9-_\.]+)

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

PreconditionFailedException

The `RevisionId` provided does not match the latest `RevisionId` for the Lambda function or alias. Call the `GetFunction` or the `GetAlias` API to retrieve the latest `RevisionId` for your resource.

HTTP Status Code: 412

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)

- [AWS SDK for Ruby V2](#)

TagResource

Adds [tags](#) to a function.

Request Syntax

```
POST /2017-03-31/tags/ARN HTTP/1.1
Content-type: application/json

{
  "Tags": {
    "string" : "string"
  }
}
```

URI Request Parameters

The request requires the following URI parameters.

Resource (p. 444)

The function's Amazon Resource Name (ARN).

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$\LATEST|[a-zA-Z0-9-_]+))?

Request Body

The request accepts the following data in JSON format.

Tags (p. 444)

A list of tags to apply to the function.

Type: String to string map

Required: Yes

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

UntagResource

Removes [tags](#) from a function.

Request Syntax

```
DELETE /2017-03-31/tags/ARN?tagKeys=TagKeys HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

[Resource \(p. 446\)](#)

The function's Amazon Resource Name (ARN).

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_+]))?

[TagKeys \(p. 446\)](#)

A list of tag keys to remove from the function.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

UpdateAlias

Updates the configuration of a Lambda function [alias](#).

Request Syntax

```
PUT /2015-03-31/functions/FunctionName/aliases/Name HTTP/1.1
Content-type: application/json

{
  "Description": "string",
  "FunctionVersion": "string",
  "RevisionId": "string",
  "RoutingConfig": {
    "AdditionalVersionWeights": {
      "string" : number
    }
  }
}
```

URI Request Parameters

The request requires the following URI parameters.

[FunctionName](#) (p. 448)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

[Name](#) (p. 448)

The name of the alias.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (?![0-9]+\\$)([a-zA-Z0-9-_]+)

Request Body

The request accepts the following data in JSON format.

[Description](#) (p. 448)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

[FunctionVersion \(p. 448\)](#)

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST|[0-9]+)

Required: No

[RevisionId \(p. 448\)](#)

Only update the alias if the revision ID matches the ID that's specified. Use this option to avoid modifying an alias that has changed since you last read it.

Type: String

Required: No

[RoutingConfig \(p. 448\)](#)

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 475\)](#) object

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "AliasArn": "string",
    "Description": "string",
    "FunctionVersion": "string",
    "Name": "string",
    "RevisionId": "string",
    "RoutingConfig": {
        "AdditionalVersionWeights": {
            "string" : number
        }
    }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[AliasArn \(p. 449\)](#)

The Amazon Resource Name (ARN) of the alias.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$LATEST|[a-zA-Z0-9-_]+))?

[Description \(p. 449\)](#)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[FunctionVersion \(p. 449\)](#)

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST|[0-9]+)

[Name \(p. 449\)](#)

The name of the alias.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (?![0-9]+\$)([a-zA-Z0-9-_]+)

[RevisionId \(p. 449\)](#)

A unique identifier that changes when you update the alias.

Type: String

[RoutingConfig \(p. 449\)](#)

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 475\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

PreconditionFailedException

The `RevisionId` provided does not match the latest `RevisionId` for the Lambda function or alias. Call the `GetFunction` or the `GetAlias` API to retrieve the latest `RevisionId` for your resource.

HTTP Status Code: 412

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

UpdateEventSourceMapping

Updates an event source mapping. You can change the function that AWS Lambda invokes, or pause invocation and resume later from the same location.

Request Syntax

```
PUT /2015-03-31/event-source-mappings/UUID HTTP/1.1
Content-type: application/json

{
  "BatchSize": number,
  "Enabled": boolean,
  "FunctionName": "string"
}
```

URI Request Parameters

The request requires the following URI parameters.

UUID (p. 452)

The identifier of the event source mapping.

Request Body

The request accepts the following data in JSON format.

BatchSize (p. 452)

The maximum number of items to retrieve in a single batch.

- **Amazon Kinesis** - Default 100. Max 10,000.
- **Amazon DynamoDB Streams** - Default 100. Max 1,000.
- **Amazon Simple Queue Service** - Default 10. Max 10.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

Required: No

Enabled (p. 452)

Disables the event source mapping to pause polling and invocation.

Type: Boolean

Required: No

FunctionName (p. 452)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.

- **Version or Alias ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction:PROD.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it's limited to 64 characters in length.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: No

Response Syntax

```
HTTP/1.1 202
Content-type: application/json

{
    "BatchSize": number,
    "EventSourceArn": "string",
    "FunctionArn": "string",
    "LastModified": number,
    "LastProcessingResult": "string",
    "State": "string",
    "StateTransitionReason": "string",
    "UUID": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 202 response.

The following data is returned in JSON format by the service.

BatchSize (p. 453)

The maximum number of items to retrieve in a single batch.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

EventSourceArn (p. 453)

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+:([a-z]{2}(-gov)?-[a-z]+\d{1}):?(\d{12}):?(.*)

FunctionArn (p. 453)

The ARN of the Lambda function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$LATEST|[a-zA-Z0-9-_]+))?

[LastModified \(p. 453\)](#)

The date that the event source mapping was last updated, in Unix time seconds.

Type: Timestamp

[LastProcessingResult \(p. 453\)](#)

The result of the last AWS Lambda invocation of your Lambda function.

Type: String

[State \(p. 453\)](#)

The state of the event source mapping. It can be one of the following: Creating, Enabling, Enabled, Disabling, Disabled, Updating, or Deleting.

Type: String

[StateTransitionReason \(p. 453\)](#)

The cause of the last state change, either User initiated or Lambda initiated.

Type: String

[UUID \(p. 453\)](#)

The identifier of the event source mapping.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the CreateFunction or the UpdateFunctionConfiguration API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceConflictException

The resource already exists.

HTTP Status Code: 409

ResourceInUseException

The operation conflicts with the resource's availability. For example, you attempted to update an EventSource Mapping in CREATING, or tried to delete a EventSource mapping currently in the UPDATING state.

HTTP Status Code: 400

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

UpdateFunctionCode

Updates a Lambda function's code.

The function's code is locked when you publish a version. You can't modify the code of a published version, only the unpublished version.

Request Syntax

```
PUT /2015-03-31/functions/FunctionName/code HTTP/1.1
Content-type: application/json

{
  "DryRun": boolean,
  "Publish": boolean,
  "RevisionId": "string",
  "S3Bucket": "string",
  "S3Key": "string",
  "S3ObjectVersion": "string",
  "ZipFile": blob
}
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 456)

The name of the Lambda function.

Name formats

- **Function name** - my-function.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Request Body

The request accepts the following data in JSON format.

DryRun (p. 456)

Set to true to validate the request parameters and access permissions without modifying the function code.

Type: Boolean

Required: No

[Publish \(p. 456\)](#)

Set to true to publish a new version of the function after updating the code. This has the same effect as calling [PublishVersion \(p. 430\)](#) separately.

Type: Boolean

Required: No

[RevisionId \(p. 456\)](#)

Only update the function if the revision ID matches the ID that's specified. Use this option to avoid modifying a function that has changed since you last read it.

Type: String

Required: No

[S3Bucket \(p. 456\)](#)

An Amazon S3 bucket in the same AWS Region as your function. The bucket can be in a different AWS account.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 63.

Pattern: ^[0-9A-Za-z\.\-_]*(\?<!\.).\$

Required: No

[S3Key \(p. 456\)](#)

The Amazon S3 key of the deployment package.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Required: No

[S3ObjectVersion \(p. 456\)](#)

For versioned objects, the version of the deployment package object to use.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Required: No

[ZipFile \(p. 456\)](#)

The base64-encoded contents of the deployment package. AWS SDK and AWS CLI clients handle the encoding for you.

Type: Base64-encoded binary data object

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json
```

```
{
  "CodeSha256": "string",
  "CodeSize": number,
  "DeadLetterConfig": {
    "TargetArn": "string"
  },
  "Description": "string",
  "Environment": {
    "Error": {
      "ErrorCode": "string",
      "Message": "string"
    },
    "Variables": {
      "string" : "string"
    }
  },
  "FunctionArn": "string",
  "FunctionName": "string",
  "Handler": "string",
  "KMSKeyArn": "string",
  "LastModified": "string",
  "Layers": [
    {
      "Arn": "string",
      "CodeSize": number
    }
  ],
  "MasterArn": "string",
  "MemorySize": number,
  "RevisionId": "string",
  "Role": "string",
  "Runtime": "string",
  "Timeout": number,
  "TracingConfig": {
    "Mode": "string"
  },
  "Version": "string",
  "VpcConfig": {
    "SecurityGroupIds": [ "string" ],
    "SubnetIds": [ "string" ],
    "VpcId": "string"
  }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

CodeSha256 (p. 457)

The SHA256 hash of the function's deployment package.

Type: String

CodeSize (p. 457)

The size of the function's deployment package, in bytes.

Type: Long

DeadLetterConfig (p. 457)

The function's dead letter queue.

Type: [DeadLetterConfig \(p. 477\)](#) object

[Description \(p. 457\)](#)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[Environment \(p. 457\)](#)

The function's environment variables.

Type: [EnvironmentResponse \(p. 480\)](#) object

[FunctionArn \(p. 457\)](#)

The function's Amazon Resource Name (ARN).

Type: String

Pattern: `arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[FunctionName \(p. 457\)](#)

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: `(arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?\d{12}:?(function:)?([a-zA-Z0-9-_\.]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[Handler \(p. 457\)](#)

The function that Lambda calls to begin executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

[KMSKeyArn \(p. 457\)](#)

The KMS key that's used to encrypt the function's environment variables. This key is only returned if you've configured a customer-managed CMK.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*):[a-zA-Z0-9-_\.]+\.*)|()`

[LastModified \(p. 457\)](#)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[Layers \(p. 457\)](#)

The function's [layers](#).

Type: Array of [Layer \(p. 489\)](#) objects

[MasterArn \(p. 457\)](#)

For Lambda@Edge functions, the ARN of the master function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[MemorySize \(p. 457\)](#)

The memory that's allocated to the function.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 3008.

[RevisionId \(p. 457\)](#)

The latest updated revision of the function or alias.

Type: String

[Role \(p. 457\)](#)

The function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\-/]+`

[Runtime \(p. 457\)](#)

The runtime environment for the Lambda function.

Type: String

Valid Values: `nodejs8.10 | nodejs10.x | java8 | python2.7 | python3.6 | python3.7 | dotnetcore1.0 | dotnetcore2.1 | go1.x | ruby2.5 | provided`

[Timeout \(p. 457\)](#)

The amount of time that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

[TracingConfig \(p. 457\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 496\)](#) object

[Version \(p. 457\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

[VpcConfig \(p. 457\)](#)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 498\)](#) object

Errors

CodeStorageExceededException

You have exceeded your maximum total code size per account. [Learn more](#)

HTTP Status Code: 400

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

PreconditionFailedException

The `RevisionId` provided does not match the latest `RevisionId` for the Lambda function or alias. Call the `GetFunction` or the `GetAlias` API to retrieve the latest `RevisionId` for your resource.

HTTP Status Code: 412

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

UpdateFunctionConfiguration

Modify the version-specific settings of a Lambda function.

These settings can vary between versions of a function and are locked when you publish a version. You can't modify the configuration of a published version, only the unpublished version.

To configure function concurrency, use [PutFunctionConcurrency \(p. 436\)](#). To grant invoke permissions to an account or AWS service, use [AddPermission \(p. 343\)](#).

Request Syntax

```
PUT /2015-03-31/functions/FunctionName/configuration HTTP/1.1
Content-type: application/json

{
  "DeadLetterConfig": {
    "TargetArn": "string"
  },
  "Description": "string",
  "Environment": {
    "Variables": {
      "string": "string"
    }
  },
  "Handler": "string",
  "KMSKeyArn": "string",
  "Layers": [ "string" ],
  "MemorySize": number,
  "RevisionId": "string",
  "Role": "string",
  "Runtime": "string",
  "Timeout": number,
  "TracingConfig": {
    "Mode": "string"
  },
  "VpcConfig": {
    "SecurityGroupIds": [ "string" ],
    "SubnetIds": [ "string" ]
  }
}
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName (p. 463)

The name of the Lambda function.

Name formats

- **Function name** - my-function.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Request Body

The request accepts the following data in JSON format.

[DeadLetterConfig \(p. 463\)](#)

A dead letter queue configuration that specifies the queue or topic where Lambda sends asynchronous events when they fail processing. For more information, see [Dead Letter Queues](#).

Type: [DeadLetterConfig \(p. 477\)](#) object

Required: No

[Description \(p. 463\)](#)

A description of the function.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

[Environment \(p. 463\)](#)

Environment variables that are accessible from function code during execution.

Type: [Environment \(p. 478\)](#) object

Required: No

[Handler \(p. 463\)](#)

The name of the method within your code that Lambda calls to execute your function. The format includes the file name. It can also include namespaces and other qualifiers, depending on the runtime. For more information, see [Programming Model](#).

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

Required: No

[KMSKeyArn \(p. 463\)](#)

The ARN of the AWS Key Management Service (AWS KMS) key that's used to encrypt your function's environment variables. If it's not provided, AWS Lambda uses a default service key.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-.]+:[^.]+|())`

Required: No

[Layers \(p. 463\)](#)

A list of [function layers](#) to add to the function's execution environment. Specify each layer by its ARN, including the version.

Type: Array of strings

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+`

Required: No

[MemorySize \(p. 463\)](#)

The amount of memory that your function has access to. Increasing the function's memory also increases its CPU allocation. The default value is 128 MB. The value must be a multiple of 64 MB.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 3008.

Required: No

[RevisionId \(p. 463\)](#)

Only update the function if the revision ID matches the ID that's specified. Use this option to avoid modifying a function that has changed since you last read it.

Type: String

Required: No

[Role \(p. 463\)](#)

The Amazon Resource Name (ARN) of the function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@/-/_/]+`

Required: No

[Runtime \(p. 463\)](#)

The identifier of the function's [runtime](#).

Type: String

Valid Values: `nodejs8.10 | nodejs10.x | java8 | python2.7 | python3.6 | python3.7 | dotnetcore1.0 | dotnetcore2.1 | go1.x | ruby2.5 | provided`

Required: No

[Timeout \(p. 463\)](#)

The amount of time that Lambda allows a function to run before stopping it. The default is 3 seconds. The maximum allowed value is 900 seconds.

Type: Integer

Valid Range: Minimum value of 1.

Required: No

[TracingConfig \(p. 463\)](#)

Set `Mode` to `Active` to sample and trace a subset of incoming requests with AWS X-Ray.

Type: [TracingConfig \(p. 495\)](#) object

Required: No

VpcConfig (p. 463)

For network connectivity to AWS resources in a VPC, specify a list of security groups and subnets in the VPC. When you connect a function to a VPC, it can only access resources and the internet through that VPC. For more information, see [VPC Settings](#).

Type: [VpcConfig \(p. 497\)](#) object

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "CodeSha256": "string",
    "CodeSize": number,
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
    "Description": "string",
    "Environment": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "Variables": {
            "string" : "string"
        }
    },
    "FunctionArn": "string",
    "FunctionName": "string",
    "Handler": "string",
    "KMSKeyArn": "string",
    "LastModified": "string",
    "Layers": [
        {
            "Arn": "string",
            "CodeSize": number
        }
    ],
    "MasterArn": "string",
    "MemorySize": number,
    "RevisionId": "string",
    "Role": "string",
    "Runtime": "string",
    "Timeout": number,
    "TracingConfig": {
        "Mode": "string"
    },
    "Version": "string",
    "VpcConfig": {
        "SecurityGroupIds": [ "string" ],
        "SubnetIds": [ "string" ],
        "VpcId": "string"
    }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

CodeSha256 (p. 466)

The SHA256 hash of the function's deployment package.

Type: String

CodeSize (p. 466)

The size of the function's deployment package, in bytes.

Type: Long

DeadLetterConfig (p. 466)

The function's dead letter queue.

Type: [DeadLetterConfig \(p. 477\)](#) object

Description (p. 466)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Environment (p. 466)

The function's environment variables.

Type: [EnvironmentResponse \(p. 480\)](#) object

FunctionArn (p. 466)

The function's Amazon Resource Name (ARN).

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:(\$\$LATEST|[a-zA-Z0-9-_]+))?`

FunctionName (p. 466)

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\$\$LATEST|[a-zA-Z0-9-_]+))?`

Handler (p. 466)

The function that Lambda calls to begin executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

KMSKeyArn (p. 466)

The KMS key that's used to encrypt the function's environment variables. This key is only returned if you've configured a customer-managed CMK.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-z0-9-.]+:[.]*|()`

[LastModified \(p. 466\)](#)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[Layers \(p. 466\)](#)

The function's [layers](#).

Type: Array of [Layer \(p. 489\)](#) objects

[MasterArn \(p. 466\)](#)

For Lambda@Edge functions, the ARN of the master function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[MemorySize \(p. 466\)](#)

The memory that's allocated to the function.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 3008.

[RevisionId \(p. 466\)](#)

The latest updated revision of the function or alias.

Type: String

[Role \(p. 466\)](#)

The function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\-_/.]+`

[Runtime \(p. 466\)](#)

The runtime environment for the Lambda function.

Type: String

Valid Values: nodejs8.10 | nodejs10.x | java8 | python2.7 | python3.6 | python3.7 | dotnetcore1.0 | dotnetcore2.1 | go1.x | ruby2.5 | provided

[Timeout \(p. 466\)](#)

The amount of time that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

[TracingConfig \(p. 466\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 496\)](#) object

[Version \(p. 466\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (`\$LATEST` | [0-9]+)

[VpcConfig \(p. 466\)](#)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 498\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

PreconditionFailedException

The `RevisionId` provided does not match the latest `RevisionId` for the Lambda function or alias. Call the `GetFunction` or the `GetAlias` API to retrieve the latest `RevisionId` for your resource.

HTTP Status Code: 412

ResourceConflictException

The resource already exists.

HTTP Status Code: 409

ResourceNotFoundException

The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

Request throughput limit exceeded.

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

Data Types

The following data types are supported:

- [AccountLimit \(p. 471\)](#)
- [AccountUsage \(p. 472\)](#)
- [AliasConfiguration \(p. 473\)](#)
- [AliasRoutingConfiguration \(p. 475\)](#)
- [Concurrency \(p. 476\)](#)
- [DeadLetterConfig \(p. 477\)](#)
- [Environment \(p. 478\)](#)
- [EnvironmentError \(p. 479\)](#)
- [EnvironmentResponse \(p. 480\)](#)
- [EventSourceMappingConfiguration \(p. 481\)](#)
- [FunctionCode \(p. 483\)](#)
- [FunctionCodeLocation \(p. 484\)](#)
- [FunctionConfiguration \(p. 485\)](#)
- [Layer \(p. 489\)](#)
- [LayersListItem \(p. 490\)](#)
- [LayerVersionContentInput \(p. 491\)](#)
- [LayerVersionContentOutput \(p. 492\)](#)
- [LayerVersionsListItem \(p. 493\)](#)
- [TracingConfig \(p. 495\)](#)
- [TracingConfigResponse \(p. 496\)](#)
- [VpcConfig \(p. 497\)](#)
- [VpcConfigResponse \(p. 498\)](#)

AccountLimit

Limits that are related to concurrency and code storage. All file and storage sizes are in bytes.

Contents

CodeSizeUnzipped

The maximum size of your function's code and layers when they're extracted.

Type: Long

Required: No

CodeSizeZipped

The maximum size of a deployment package when it's uploaded directly to AWS Lambda. Use Amazon S3 for larger files.

Type: Long

Required: No

ConcurrentExecutions

The maximum number of simultaneous function executions.

Type: Integer

Required: No

TotalCodeSize

The amount of storage space that you can use for all deployment packages and layer archives.

Type: Long

Required: No

UnreservedConcurrentExecutions

The maximum number of simultaneous function executions, minus the capacity that's reserved for individual functions with [PutFunctionConcurrency](#) (p. 436).

Type: Integer

Valid Range: Minimum value of 0.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

AccountUsage

The number of functions and amount of storage in use.

Contents

FunctionCount

The number of Lambda functions.

Type: Long

Required: No

TotalCodeSize

The amount of storage space, in bytes, that's being used by deployment packages and layer archives.

Type: Long

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

AliasConfiguration

Provides configuration information about a Lambda function [alias](#).

Contents

AliasArn

The Amazon Resource Name (ARN) of the alias.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$\$LATEST|[a-zA-Z0-9-_]+))?`

Required: No

Description

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

FunctionVersion

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$\$LATEST|[0-9]+)`

Required: No

Name

The name of the alias.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(?!^[\0-9]+\$)([a-zA-Z0-9-_]+)`

Required: No

RevisionId

A unique identifier that changes when you update the alias.

Type: String

Required: No

RoutingConfig

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 475\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

AliasRoutingConfiguration

The [traffic-shifting](#) configuration of a Lambda function alias.

Contents

AdditionalVersionWeights

The name of the second alias, and the percentage of traffic that's routed to it.

Type: String to double map

Key Length Constraints: Minimum length of 1. Maximum length of 1024.

Key Pattern: [0-9]+

Valid Range: Minimum value of 0.0. Maximum value of 1.0.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

Concurrency

Contents

ReservedConcurrentExecutions

The number of concurrent executions that are reserved for this function. For more information, see [Managing Concurrency](#).

Type: Integer

Valid Range: Minimum value of 0.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

DeadLetterConfig

The [dead letter queue](#) for failed asynchronous invocations.

Contents

TargetArn

The Amazon Resource Name (ARN) of an Amazon SQS queue or Amazon SNS topic.

Type: String

Pattern: (`arn:(aws[a-zA-Z-]*):[a-z0-9-.]+:[.*]|()`)

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

Environment

A function's environment variable settings.

Contents

Variables

Environment variable key-value pairs.

Type: String to string map

Key Pattern: [a-zA-Z]([a-zA-Z0-9_])⁺

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

EnvironmentError

Error messages for environment variables that couldn't be applied.

Contents

ErrorCode

The error code.

Type: String

Required: No

Message

The error message.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

EnvironmentResponse

The results of a configuration update that applied environment variables.

Contents

Error

Error messages for environment variables that couldn't be applied.

Type: [EnvironmentError \(p. 479\)](#) object

Required: No

Variables

Environment variable key-value pairs.

Type: String to string map

Key Pattern: [a-zA-Z]([a-zA-Z0-9_])⁺

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

EventSourceMappingConfiguration

A mapping between an AWS resource and an AWS Lambda function. See [CreateEventSourceMapping \(p. 351\)](#) for details.

Contents

BatchSize

The maximum number of items to retrieve in a single batch.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

Required: No

EventSourceArn

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-]+)([a-z]{2}(-gov)?-[a-z]+\-\d{1})?:(\d{12})?:(.*?)`

Required: No

FunctionArn

The ARN of the Lambda function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+\-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: No

LastModified

The date that the event source mapping was last updated, in Unix time seconds.

Type: Timestamp

Required: No

LastProcessingResult

The result of the last AWS Lambda invocation of your Lambda function.

Type: String

Required: No

State

The state of the event source mapping. It can be one of the following: `Creating`, `Enabling`, `Enabled`, `Disabling`, `Disabled`, `Updating`, or `Deleting`.

Type: String

Required: No

StateTransitionReason

The cause of the last state change, either `User` initiated or `Lambda` initiated.

Type: String

Required: No

UUID

The identifier of the event source mapping.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

FunctionCode

The code for the Lambda function. You can specify either an object in Amazon S3, or upload a deployment package directly.

Contents

S3Bucket

An Amazon S3 bucket in the same AWS Region as your function. The bucket can be in a different AWS account.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 63.

Pattern: ^[0-9A-Za-z\.\-_]*(\?<!\.).\$

Required: No

S3Key

The Amazon S3 key of the deployment package.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Required: No

S3ObjectVersion

For versioned objects, the version of the deployment package object to use.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Required: No

ZipFile

The base64-encoded contents of the deployment package. AWS SDK and AWS CLI clients handle the encoding for you.

Type: Base64-encoded binary data object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

FunctionCodeLocation

Details about a function's deployment package.

Contents

Location

A presigned URL that you can use to download the deployment package.

Type: String

Required: No

RepositoryType

The service that's hosting the file.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

FunctionConfiguration

Details about a function's configuration.

Contents

CodeSha256

The SHA256 hash of the function's deployment package.

Type: String

Required: No

CodeSize

The size of the function's deployment package, in bytes.

Type: Long

Required: No

DeadLetterConfig

The function's dead letter queue.

Type: [DeadLetterConfig \(p. 477\)](#) object

Required: No

Description

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

Environment

The function's environment variables.

Type: [EnvironmentResponse \(p. 480\)](#) object

Required: No

FunctionArn

The function's Amazon Resource Name (ARN).

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:(\$LATEST|[a-zA-Z0-9-_]+))?

Required: No

FunctionName

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: No

Handler

The function that Lambda calls to begin executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

Required: No

KMSKeyArn

The KMS key that's used to encrypt the function's environment variables. This key is only returned if you've configured a customer-managed CMK.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-.]+\.:*)|()`

Required: No

LastModified

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Required: No

Layers

The function's [layers](#).

Type: Array of [Layer \(p. 489\)](#) objects

Required: No

MasterArn

For Lambda@Edge functions, the ARN of the master function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-zA-Z]{2}(-gov)?-[a-zA-Z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: No

MemorySize

The memory that's allocated to the function.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 3008.

Required: No

RevisionId

The latest updated revision of the function or alias.

Type: String

Required: No

Role

The function's execution role.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/[a-zA-Z_0-9+=,.@\\-_/+]

Required: No

Runtime

The runtime environment for the Lambda function.

Type: String

Valid Values: nodejs8.10 | nodejs10.x | java8 | python2.7 | python3.6 | python3.7 | dotnetcore1.0 | dotnetcore2.1 | go1.x | ruby2.5 | provided

Required: No

Timeout

The amount of time that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

Required: No

TracingConfig

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 496\)](#) object

Required: No

Version

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST|[0-9]+)

Required: No

VpcConfig

The function's networking configuration.

Type: [VpcConfigResponse \(p. 498\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

Layer

An [AWS Lambda layer](#).

Contents

Arn

The Amazon Resource Name (ARN) of the function layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+`

Required: No

CodeSize

The size of the layer archive in bytes.

Type: Long

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

LayersListItem

Details about an [AWS Lambda layer](#).

Contents

LatestMatchingVersion

The newest version of the layer.

Type: [LayerVersionsListItem \(p. 493\)](#) object

Required: No

LayerArn

The Amazon Resource Name (ARN) of the function layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+`

Required: No

LayerName

The name of the layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+)|[a-zA-Z0-9-_]+`

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

LayerVersionContentInput

A ZIP archive that contains the contents of an [AWS Lambda layer](#). You can specify either an Amazon S3 location, or upload a layer archive directly.

Contents

S3Bucket

The Amazon S3 bucket of the layer archive.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 63.

Pattern: ^[0-9A-Za-z\.\-_]*(\?!\.)\$

Required: No

S3Key

The Amazon S3 key of the layer archive.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Required: No

S3ObjectVersion

For versioned objects, the version of the layer archive object to use.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Required: No

ZipFile

The base64-encoded contents of the layer archive. AWS SDK and AWS CLI clients handle the encoding for you.

Type: Base64-encoded binary data object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

LayerVersionContentOutput

Details about a version of an [AWS Lambda layer](#).

Contents

CodeSha256

The SHA-256 hash of the layer archive.

Type: String

Required: No

CodeSize

The size of the layer archive in bytes.

Type: Long

Required: No

Location

A link to the layer archive in Amazon S3 that is valid for 10 minutes.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

LayerVersionsListItem

Details about a version of an [AWS Lambda layer](#).

Contents

CompatibleRuntimes

The layer's compatible runtimes.

Type: Array of strings

Array Members: Maximum number of 5 items.

Valid Values: nodejs8.10 | nodejs10.x | java8 | python2.7 | python3.6 | python3.7 | dotnetcore1.0 | dotnetcore2.1 | go1.x | ruby2.5 | provided

Required: No

CreatedDate

The date that the version was created, in ISO 8601 format. For example, 2018-11-27T15:10:45.123+0000.

Type: String

Required: No

Description

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

LayerVersionArn

The ARN of the layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+

Required: No

LicenseInfo

The layer's open-source license.

Type: String

Length Constraints: Maximum length of 512.

Required: No

Version

The version number.

Type: Long

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

TracingConfig

The function's AWS X-Ray tracing configuration.

Contents

Mode

The tracing mode.

Type: String

Valid Values: Active | PassThrough

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

TracingConfigResponse

The function's AWS X-Ray tracing configuration.

Contents

Mode

The tracing mode.

Type: String

Valid Values: Active | PassThrough

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

VpcConfig

The VPC security groups and subnets that are attached to a Lambda function.

Contents

SecurityGroupIds

A list of VPC security groups IDs.

Type: Array of strings

Array Members: Maximum number of 5 items.

Required: No

SubnetIds

A list of VPC subnet IDs.

Type: Array of strings

Array Members: Maximum number of 16 items.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

VpcConfigResponse

The VPC security groups and subnets that are attached to a Lambda function.

Contents

SecurityGroupIds

A list of VPC security groups IDs.

Type: Array of strings

Array Members: Maximum number of 5 items.

Required: No

SubnetIds

A list of VPC subnet IDs.

Type: Array of strings

Array Members: Maximum number of 16 items.

Required: No

VpcId

The ID of the VPC.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Go - Pilot](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

Certificate Errors When Using an SDK

Because AWS SDKs use the CA certificates from your computer, changes to the certificates on the AWS servers can cause connection failures when you attempt to use an SDK. You can prevent these failures by keeping your computer's CA certificates and operating system up-to-date. If you encounter this issue in a corporate environment and do not manage your own computer, you might need to ask an administrator to assist with the update process. The following list shows minimum operating system and Java versions:

- Microsoft Windows versions that have updates from January 2005 or later installed contain at least one of the required CAs in their trust list.
- Mac OS X 10.4 with Java for Mac OS X 10.4 Release 5 (February 2007), Mac OS X 10.5 (October 2007), and later versions contain at least one of the required CAs in their trust list.

- Red Hat Enterprise Linux 5 (March 2007), 6, and 7 and CentOS 5, 6, and 7 all contain at least one of the required CAs in their default trusted CA list.
- Java 1.4.2_12 (May 2006), 5 Update 2 (March 2005), and all later versions, including Java 6 (December 2006), 7, and 8, contain at least one of the required CAs in their default trusted CA list.

When accessing the AWS Lambda management console or AWS Lambda API endpoints, whether through browsers or programmatically, you will need to ensure your client machines support any of the following CAs:

- Amazon Root CA 1
- Starfield Services Root Certificate Authority - G2
- Starfield Class 2 Certification Authority

Root certificates from the first two authorities are available from [Amazon Trust Services](#), but keeping your computer up-to-date is the more straightforward solution. To learn more about ACM-provided certificates, see [AWS Certificate Manager FAQs](#).

AWS Lambda Releases

The following table describes the important changes to the *AWS Lambda Developer Guide* after May 2018. For notification about updates to this documentation, subscribe to the [RSS feed](#).

update-history-change	update-history-description	update-history-date
Node.js 10	A new runtime is available for Node.js 10, nodejs10.x. This runtime uses Node.js 10.15 and will be updated with the latest point release of Node.js 10 periodically. Node.js 10 is also the first runtime to use Amazon Linux 2. See Building Lambda Functions with Node.js for details.	May 9, 2019
GetLayerVersionByArn API	Use the GetLayerVersionByArn API to download layer version information with the version ARN as input. Compared to GetLayerVersion, GetLayerVersionByArn lets you use the ARN directly instead of parsing it to get the layer name and version number.	April 25, 2019
Ruby	AWS Lambda now supports Ruby 2.5 with a new runtime. See Building Lambda Functions with Ruby for details.	November 29, 2018
Layers	With Lambda layers, you can package and deploy libraries, custom runtimes, and other dependencies separately from your function code. Share your layers with your other accounts or the whole world. See AWS Lambda Layers for details.	November 29, 2018
Custom Runtimes	Build a custom runtime to run Lambda functions in your favorite programming language. See Custom AWS Lambda Runtimes for details.	November 29, 2018
Application Load Balancer Triggers	Elastic Load Balancing now supports Lambda functions as a target for Application Load Balancers. See Using Lambda with Application Load Balancers for details.	November 29, 2018

Use Kinesis HTTP/2 Stream Consumers as a Trigger	You can use Kinesis HTTP/2 data stream consumers to send events to AWS Lambda. Stream consumers have dedicated read throughput from each shard in your data stream and use HTTP/2 to minimize latency. See Using AWS Lambda with Kinesis for details.	November 19, 2018
Python 3.7	AWS Lambda now supports Python 3.7 with a new runtime. For more information, see Building Lambda Functions with Python .	November 19, 2018
Asynchronous function invocation payload limit increase	The maximum payload size for asynchronous invocations increased from 128 KB to 256 KB, matching the maximum message size from an Amazon SNS trigger. See AWS Lambda Limits for details.	November 16, 2018
AWS GovCloud (US-East) Region	AWS Lambda is now available in the AWS GovCloud (US-East) region. For details, see AWS GovCloud (US-East) Now Open on the AWS blog.	November 12, 2018
Moved AWS SAM topics to a separate Developer Guide	A number of topics were focused on building serverless applications using the AWS Serverless Application Model (AWS SAM). These topics have been moved to AWS Serverless Application Model Developer Guide .	October 25, 2018
View Lambda applications in the console	You can view the status of your Lambda applications on the Applications page in the Lambda console. This page shows the status of the AWS CloudFormation stack. It includes links to pages where you can view more information about the resources in the stack. You can also view aggregate metrics for the application and create custom monitoring dashboards.	October 11, 2018

Function execution timeout limit	To allow for long-running functions, the maximum configurable execution timeout increased from 5 minutes to 15 minutes. See AWS Lambda Limits for details.	October 10, 2018
Support for PowerShell Core language in AWS Lambda	AWS Lambda now supports the PowerShell Core language. For more information, see Programming Model for Authoring Lambda Functions in PowerShell .	September 11, 2018
Support for .NET Core 2.1.0 runtime in AWS Lambda	AWS Lambda now supports the .NET Core 2.1.0 runtime. For more information, see .NET Core CLI .	July 9, 2018
Updates now available over RSS	You can now subscribe to an RSS feed to receive notifications to the <i>AWS Lambda Developer Guide</i> .	July 5, 2018
Support for Amazon SQS as event source	AWS Lambda now supports Amazon Simple Queue Service (Amazon SQS) as an event source. For more information, see Invoking Lambda Functions .	June 28, 2018
China (Ningxia) Region	AWS Lambda is now available in the China (Ningxia) Region. For more information about Lambda regions and endpoints, see Regions and Endpoints in the <i>AWS General Reference</i> .	June 28, 2018

Earlier Updates

The following table describes the important changes in each release of the *AWS Lambda Developer Guide* before June 2018.

Change	Description	Date
Runtime support for Node.js runtime 8.10	AWS Lambda now supports Node.js runtime version 8.10. For more information, see Building Lambda Functions with Node.js (p. 241) .	April 2, 2018
Function and alias revision IDs	AWS Lambda now supports revision IDs on your function versions and aliases. You can use these IDs to track and apply conditional updates when you are updating your function version or alias resources.	January 25, 2018
Runtime support for Go and .NET 2.0	AWS Lambda has added runtime support for Go and .NET 2.0. For more information, see Building Lambda Functions with Go (p. 290) and Building Lambda Functions with C# (p. 302) .	January 15, 2018

Change	Description	Date
Console Redesign	AWS Lambda has introduced a new Lambda console to simplify your experience and added a Cloud9 Code Editor to enhance your ability debug and revise your function code. For more information, see Creating Functions Using the AWS Lambda Console Editor (p. 24) .	November 30, 2017
Setting Concurrency Limits on Individual Functions	AWS Lambda now supports setting concurrency limits on individual functions. For more information, see Managing Concurrency (p. 37) .	November 30, 2017
Shifting Traffic with Aliases	AWS Lambda now supports shifting traffic with aliases. For more information, see Traffic Shifting Using Aliases (p. 62) .	November 28, 2017
Gradual Code Deployment	AWS Lambda now supports safely deploying new versions of your Lambda function by leveraging Code Deploy. For more information, see Gradual Code Deployment .	November 28, 2017
China (Beijing) Region	AWS Lambda is now available in the China (Beijing) Region. For more information about Lambda regions and endpoints, see Regions and Endpoints in the <i>AWS General Reference</i> .	November 9, 2017
Introducing SAM Local	AWS Lambda introduces SAM Local (now known as SAM CLI), a AWS CLI tool that provides an environment for you to develop, test, and analyze your serverless applications locally before uploading them to the Lambda runtime. For more information, see Testing and Debugging Serverless Applications .	August 11, 2017
Canada (Central) Region	AWS Lambda is now available in the Canada (Central) Region. For more information about Lambda regions and endpoints, see Regions and Endpoints in the <i>AWS General Reference</i> .	June 22, 2017
South America (São Paulo) Region	AWS Lambda is now available in the South America (São Paulo) Region. For more information about Lambda regions and endpoints, see Regions and Endpoints in the <i>AWS General Reference</i> .	June 6, 2017
AWS Lambda support for AWS X-Ray.	Lambda introduces support for X-Ray, which allows you to detect, analyze, and optimize performance issues with your Lambda applications. For more information, see Using AWS X-Ray (p. 232) .	April 19, 2017
Asia Pacific (Mumbai) Region	AWS Lambda is now available in the Asia Pacific (Mumbai) Region. For more information about Lambda regions and endpoints, see Regions and Endpoints in the <i>AWS General Reference</i> .	March 28, 2017
AWS Lambda now supports Node.js runtime v6.10	AWS Lambda added support for Node.js runtime v6.10. For more information, see Building Lambda Functions with Node.js (p. 241) .	March 22, 2017
EU (London) Region	AWS Lambda is now available in the EU (London) Region. For more information about Lambda regions and endpoints, see Regions and Endpoints in the <i>AWS General Reference</i> .	February 1, 2017

Change	Description	Date
AWS Lambda support for the .NET runtime, Lambda@Edge (Preview), Dead Letter Queues and automated deployment of serverless applications.	<p>AWS Lambda added support for C#. For more information, see Building Lambda Functions with C# (p. 302).</p> <p>Lambda@Edge allows you to run Lambda functions at the AWS Edge locations in response to CloudFront events. For more information, see Using AWS Lambda with CloudFront Lambda@Edge (p. 166).</p>	December 3, 2016
AWS Lambda adds Amazon Lex as a supported event source.	Using Lambda and Amazon Lex, you can quickly build chat bots for various services like Slack and Facebook. For more information, see Using AWS Lambda with Amazon Lex (p. 193) .	November 30, 2016
US West (N. California) Region	AWS Lambda is now available in the US West (N. California) Region. For more information about Lambda regions and endpoints, see Regions and Endpoints in the <i>AWS General Reference</i> .	November 21, 2016
Introduced the AWS Serverless Application Model for creating and deploying Lambda-based applications and using environment variables for Lambda function configuration settings.	<p>AWS Serverless Application Model: You can now use the AWS SAM to define the syntax for expressing resources within a serverless application. In order to deploy your application, simply specify the resources you need as part of your application, along with their associated permissions policies in a AWS CloudFormation template file (written in either JSON or YAML), package your deployment artifacts, and deploy the template. For more information, see AWS Lambda Applications (p. 116).</p> <p>Environment variables: You can use environment variables to specify configuration settings for your Lambda function outside of your function code. For more information, see AWS Lambda Environment Variables (p. 40).</p>	November 18, 2016
Asia Pacific (Seoul) Region	AWS Lambda is now available in the Asia Pacific (Seoul) Region. For more information about Lambda regions and endpoints, see Regions and Endpoints in the <i>AWS General Reference</i> .	August 29, 2016
Asia Pacific (Sydney) Region	Lambda is now available in the Asia Pacific (Sydney) Region. For more information about Lambda regions and endpoints, see Regions and Endpoints in the <i>AWS General Reference</i> .	June 23, 2016
Updates to the Lambda console	The Lambda console has been updated to simplify the role-creation process. For more information, see Create a Lambda Function with the Console (p. 3) .	June 23, 2016
AWS Lambda now supports Node.js runtime v4.3	AWS Lambda added support for Node.js runtime v4.3. For more information, see Building Lambda Functions with Node.js (p. 241) .	April 07, 2016
EU (Frankfurt) region	Lambda is now available in the EU (Frankfurt) region. For more information about Lambda regions and endpoints, see Regions and Endpoints in the <i>AWS General Reference</i> .	March 14, 2016
VPC support	You can now configure a Lambda function to access resources in your VPC. For more information, see Configuring a Lambda Function to Access Resources in an Amazon VPC (p. 68) .	February 11, 2016

Change	Description	Date
AWS Lambda runtime has been updated.	The execution environment (p. 102) has been updated.	November 4, 2015
Versioning support, Python for developing code for Lambda functions, scheduled events, and increase in execution time	<p>You can now develop your Lambda function code using Python. For more information, see Building Lambda Functions with Python (p. 251).</p> <p>Versioning: You can maintain one or more versions of your Lambda function. Versioning allows you to control which Lambda function version is executed in different environments (for example, development, testing, or production). For more information, see AWS Lambda Function Versioning and Aliases (p. 47).</p> <p>Scheduled events: You can also set up AWS Lambda to invoke your code on a regular, scheduled basis using the AWS Lambda console. You can specify a fixed rate (number of hours, days, or weeks) or you can specify a cron expression. For an example, see Using AWS Lambda with Amazon CloudWatch Events (p. 158).</p> <p>Increase in execution time: You can now set up your Lambda functions to run for up to five minutes allowing longer running functions such as large volume data ingestion and processing jobs.</p>	October 08, 2015
Support for DynamoDB Streams	DynamoDB Streams is now generally available and you can use it in all the regions where DynamoDB is available. You can enable DynamoDB Streams for your table and use a Lambda function as a trigger for the table. Triggers are custom actions you take in response to updates made to the DynamoDB table. For an example walkthrough, see Tutorial: Using AWS Lambda with Amazon DynamoDB Streams (p. 172) .	July 14, 2015
AWS Lambda now supports invoking Lambda functions with REST-compatible clients.	<p>Until now, to invoke your Lambda function from your web, mobile, or IoT application you needed the AWS SDKs (for example, AWS SDK for Java, AWS SDK for Android, or AWS SDK for iOS). Now, AWS Lambda supports invoking a Lambda function with REST-compatible clients through a customized API that you can create using Amazon API Gateway. You can send requests to your Lambda function endpoint URL. You can configure security on the endpoint to allow open access, leverage AWS Identity and Access Management (IAM) to authorize access, or use API keys to meter access to your Lambda functions by others.</p> <p>For an example Getting Started exercise, see Using AWS Lambda with Amazon API Gateway (p. 135).</p> <p>For more information about the Amazon API Gateway, see https://aws.amazon.com/api-gateway/.</p>	July 09, 2015

Change	Description	Date
The AWS Lambda console now provides blueprints to easily create Lambda functions and test them.	AWS Lambda console provides a set of <i>blueprints</i> . Each blueprint provides a sample event source configuration and sample code for your Lambda function that you can use to easily create Lambda-based applications. All of the AWS Lambda Getting Started exercises now use the blueprints. For more information, see Getting Started with AWS Lambda (p. 3) .	In this release
AWS Lambda now supports Java to author your Lambda functions.	You can now author Lambda code in Java. For more information, see Building Lambda Functions with Java (p. 263) .	June 15, 2015
AWS Lambda now supports specifying an Amazon S3 object as the function .zip when creating or updating a Lambda function.	You can upload a Lambda function deployment package (.zip file) to an Amazon S3 bucket in the same region where you want to create a Lambda function. Then, you can specify the bucket name and object key name when you create or update a Lambda function.	May 28, 2015
AWS Lambda now generally available with added support for mobile backends	<p>AWS Lambda is now generally available for production use. The release also introduces new features that make it easier to build mobile, tablet, and Internet of Things (IoT) backends using AWS Lambda that scale automatically without provisioning or managing infrastructure. AWS Lambda now supports both real-time (synchronous) and asynchronous events. Additional features include easier event source configuration and management. The permission model and the programming model have been simplified by the introduction of resource policies for your Lambda functions.</p> <p>The documentation has been updated accordingly. For information, see the following topics:</p> <p>Getting Started with AWS Lambda (p. 3)</p> <p>AWS Lambda</p>	April 9, 2015
Preview release	Preview release of the <i>AWS Lambda Developer Guide</i> .	November 13, 2014

AWS Glossary

For the latest AWS terminology, see the [AWS Glossary](#) in the *AWS General Reference*.