Serverless Compute

NOTE: This content will be available as a video course soon! We are making it available as a Learning Path resource in the meantime to help you prepare for the newly updated AWS Certified Solutions Architect - Associate (SAA-C03) exam.

Introduction

Hello and welcome to this course, which will provide an overview of AWS Lambda and how to get started using the service.

Before we begin, I'd like to introduce myself - my name is Alana Layton and I'm an AWS content creator here at Cloud Academy. Feel free to connect with me to ask any questions using the details shown on the screen. Alternatively you can always get in touch with us here at Cloud Academy by sending an e-mail to support@cloudacademy.com where one of our Cloud experts will reply to your question.

Who should attend this course?

This course has been created for those that are looking to get started using AWS Lambda, or for those looking for information about Lambda for an AWS certification.

Learning Objectives

By the end of this course, you should have a greater understanding of:

- What AWS Lambda is and its benefits,
- How to use AWS Lambda,
- And how to monitor and troubleshoot the service

Prerequisites

To get the most out of this course, you should have a fundamental understanding of AWS, including some of the foundational services such as Amazon EC2, Amazon S3, Amazon CloudWatch, and AWS IAM. For more information on these services, check out the following courses titled:

An Introduction to Amazon S3
https://cloudacademy.com/course/introduction-to-amazon-s3/

Compute Fundamentals for AWS
https://cloudacademy.com/course/compute-fundamentals-for-aws/

Managing Access using IAM User Groups and Roles
https://cloudacademy.com/course/managing-access-using-iam-user-groups-roles-2175/

An Overview of Amazon CloudWatch
https://cloudacademy.com/course/an-overview-of-amazon-cloudwatch-1222/

Feedback

Feedback on our courses here at Cloud Academy is valuable to both us as trainers and any students looking to take the same course in the future. If you have any feedback, positive or negative, it would be greatly appreciated if you could contact support@cloudacademy.com.

Please note that, at the time of writing this content, all course information was accurate. AWS implements hundreds of updates every month as part of its ongoing drive to innovate and enhance its services.

As a result, minor discrepancies may appear in the course content over time. Here at Cloud Academy, we strive to keep our content up to date in order to provide the best training available. So, if you notice any information that is outdated, please contact support@cloudacademy.com. This will allow us to update the course during its next release cycle.

Thank you!

An Overview of AWS Lambda

To really understand serverless compute, you have to first understand servers. For example, I want you to think about all the work that goes into running an EC2 instance: you have to install software, patch the instance, manage scaling and high availability, configure storage, and then write your code for your application and deploy it to the instance.

Now think about all that infrastructure maintenance and administration going away - enabling you to focus solely on your code and business logic. That's the idea behind serverless. Now of course, this maintenance and server administration still exists behind the scenes, however, it's no longer your job to do it - it becomes the service's responsibility

The serverless compute service we'll focus on in this course is called AWS Lambda. Understanding Lambda is the same as understanding almost any function in a piece of code. There are three major parts:

1. The input
2. The function, and
3. The output.

The Function

Let's start with the function. Just as EC2 is made up of instances, Lambda is made up of functions. Functions are the code that you write that represents your business logic. In this function you also configure other important details, such as permissions, environment variables, and the amount of power the function needs. The way you specify power is by choosing how much memory you want to allocate to your function. The service then uses this number and provides proportional amounts of CPU, network and disk I/O.

To upload your code to the service, you can either write the code directly in the service itself or you can upload this code via a zip file or files stored in Amazon S3. The programming language you write your code in must match the runtime you select in the service.

There are several options for runtimes. You can use a runtime that lambda natively supports, such as:

- Java,
- Go,
- Powershell,
- Node.js,
- C#,
- Python, and
- Ruby.

Or, if you want to use a language that isn't in that list, you can choose to bring other languages by using the custom runtime API. So if you're thrilled at the idea of running PHP or C in Lambda, the custom runtime feature is for you.

The Input

Once you've uploaded or written your code - how does your code run? Well, it has to be invoked. This is where the first piece of the equation - the input - comes into play. There are several options for your function to be invoked:

- Your function could be invoked directly through the console, SDK, AWS toolkits, or through the CLI.

- It could be invoked using a function URL, which is an HTTP endpoint you can enable for your Lambda function
- Or it could be invoked automatically by a trigger, such as an AWS service or resource. These triggers will run your function in response to certain events or on a schedule that you specify. So, if you want to run your function every day at 8 am, you can do that.

When you invoke your function, you can pass in events for the function to process. If a service invokes your function, they can also pass in events - however, the service will be responsible for structuring those events. For example, your code could run in response to a request from API Gateway or an S3 event, such as a PUT object API call. So once the PUT object API call is made, AWS Lambda will run your code, just as you've written it, using only the compute power you defined.

The Output

Then you have the third and final piece, which is the output. Once the function is triggered, and your code runs, your Lambda function can then make calls to downstream resources. This means that from your code, you can make API calls to other services like Amazon DynamoDB, Amazon SQS, Amazon SNS and more.

When your Lambda function is triggered, the service automatically monitors your function through logs and metrics. You can additionally choose to write custom logging statements in your code that will help you identify if your code is operating as expected. These log streams act as a recording of the sequence of events from your function. Lambda also sends common metrics of your functions to CloudWatch for monitoring and alerting.

Costs

So what do you pay for with this service? You only pay for what you use - which means three things.

1. You are charged for the amount of requests that you send to your function.
2. The Lambda function begins charging you when it is triggered, and stops charging you when the code has been executed. Otherwise known as the duration it runs. This is rounded up to the nearest 1 millisecond of use.
3. You're charged based on the amount of compute power you provision for your function. So if you provision the maximum amount of memory, you'll be charged for that.

AWS Lambda Important Configurations

In this video, let's take a look at some of the other features that are important to understand when working with AWS Lambda.

Architecture

You can choose between an x86 architecture and the arm64 architecture that uses Graviton2 processors.

The Handler function

The handler is the entrypoint of your code. The handler is made up of the combination of the file name and the function name. If you named your file anything other than lambda_function or if the function name is anything other than lambda_handler, you'll want to update the Lambda setting to match, so that Lambda can properly run your code.

Memory

You can change the amount of memory that your function is allocated. AWS uses the memory setting that you configure to calculate the CPU power allocated to the function. When you give your function more memory, you get more CPU, and up to a point more networking bandwidth as well. Currently, the smallest amount you can allocate is 128 MB and the largest is 10,240 MB.

Temporary Storage

You can also configure temporary storage space for your function. This is really helpful in data analytics workloads as you can use temporary storage to quickly process small amounts of data.

Timeout

The timeout determines how long the function should run before it terminates. The default timeout is 3 seconds, however you can increase this to 15 minutes at the maximum if needed.

Permissions

You can specify an execution IAM role for Lambda to assume..Keep in mind that whatever role you choose should have access to CloudWatch Logs, so that it can properly create log groups for your functions.

You can also create resource-based policies for your function. Resource-based policies let you grant permissions to other AWS accounts or organizations so that they can invoke or manage your function. You also use a resource-based policy to allow an AWS service to invoke your function on your behalf.

Environment Variables

Environment variables are key value pairs that allow you to incorporate variables into your function without embedding them directly into your code. Your code references these variables, which you can update at any time. So for example, you would normally want to test your function prior to moving it into the production environment. By using variables this is very easy to do. For example, let’s say your test and production environment uses a different S3 bucket. You can enter “S3Bucket” as the key, and as the value you can put in “testbucket”.

Now if the tests are successful then you could simply change the value in the variable without changing the code, allowing you to keep the code logic exactly the same. So when moving your function into production, you would simply change the variable from” testbucket” to, let's say “productionbucket”, and it's as simple as that.

You will notice that the environment variable section also has an encryption configuration option. Now by default, AWS Lambda encrypts environment variables after the function has been deployed using the AWS Lambda master key within that region via KMS, which is the Key Management Service. This data is then decrypted every time your function is invoked. However, if you are storing sensitive information within your variables it is recommended you encrypt this data prior to deployment using this enable helpers for encryption in transit checkbox.

Tags

Just like with other resources in AWS, you can create tags to help identify and group your resources.

Network

Moving on, there is the network section - By default, all of the functions you configure are run in a VPC that is controlled by AWS. However, you can choose to run your functions in your own VPC if you’d like. Here you can choose the VPC, subnets, and security groups to connect your function to. It’s recommended to connect your function to private subnets for access to private resources and use NAT or network access translation to provide internet access.

Concurrency

Then, last is the concurrency section. Any time you invoke a function, it can only process one event or request at one time. That means if you get a second event that comes while the function is still processing the first event, the service will spin up a second instance of your function to handle that request. This is referred to as concurrency. It’s how many instances of your function are spun up to handle the amount of requests that your function is serving at one time. With the concurrency controls, you can configure both reserved concurrency and provisioned concurrency.

Reserved concurrency enables you to specify the maximum number of concurrent instances of your function that you would like spun up at any given time. Once you reserve that number, no other function can use this concurrency. Without reserved concurrency, other functions in that same region can use up all of the available concurrency. This also ensures that your function doesn’t become out of control - you can cap it at a particular number so it doesn’t steal from other functions. Note that this is free to use.

Provisioned concurrency on the other hand, was created to stop the dreaded cold start process. A cold start generally occurs when the Lambda function needs to be initialized - this is most commonly seen the first time a lambda function serves a request after it gets deployed. Often this initialization period can tack on additional time for that function to finish running. That means the first time you run a function, it could take the function 500 ms to run instead of the usual 400 ms it takes after initialization.

To solve this problem, AWS came up with provisioned concurrency. It's designed to keep functions already initialized and "warm". However, it is not free to use this feature.

Invoking a Lambda function

There are several ways to invoke a Lambda function. You can invoke it directly using the console, the CLI, SDKs, or you can invoke it automatically by using a trigger such as an AWS service.

No matter how you invoke a Lambda function, even if you're invoking it directly through the Lambda service itself, you're using the service's API. Every invocation goes through the Lambda API.

What this API provides to you is three different models of how you can invoke your function:

Synchronous (Push-Based) Model

The first choice is the synchronous or push-based model. This follows the request/response model. For example, let's say we have a service like API Gateway that gets a request from a client. API Gateway then sends that request to a backend, in this case Lambda. API Gateway does this by making an invoke call to that function. After the Lambda function executes, it then returns a response back to API Gateway, which returns the response to the client. Request goes out, response comes in.

For synchronous invocations, if the function fails, the trigger is responsible for retrying it. In some cases, this might mean that there are no retries. For example, with API Gateway, it can send the error message back to the client.

To invoke a Lambda function synchronously, you can set the invocation type using the AWS SDK or you can use the CLI invoke command to do this. For example, I can use the command

```
aws lambda invoke --function-name my-function --cli-binary-format
raw-in-base64-out --payload '{ "key": "value" }' response.json
```

You can also use the –invocation-type parameter and set it as RequestResponse to invoke it synchronously.

```
aws lambda invoke --function-name my-function    --invocation-type
RequestResponse    --cli-binary-format raw-in-base64-out  --payload
'{ "key": "value" }' response.json
```

Asynchronous (Event-based) Model

The second model is the asynchronous model, also called the event-based model. In this configuration, the response does not go back to the original service that invoked the lambda function. In fact there's no path back up to the service that triggered the function to run, unless you write that logic yourself.

For example, this is common with Amazon S3 and Amazon Simple Notification Service. Say you want to trigger a lambda function to run once an object is placed in a bucket. You can do this, but it's not going to send a response back to S3 once it finishes executing, without additional business logic. The nice thing about the asynchronous model, is that it handles retries if the function returns an error or is throttled. It also uses a built-in queue. So, any event sent to your function is placed on this queue first and then eventually sent to the function.

If any of these events can't be processed for whatever reason, you can send that failed event to a dead letter queue or use Lambda destinations to send a record of the invocation to a service. The dead letter queue will receive only the content of the event, while using Lambda destinations records will include both the request context and payload as well as the response context and payload. While both are a great way to troubleshoot failed events, Destinations is the more feature rich option.

To invoke a lambda function asynchronously, I can use the invoke command, except this time, I would need to use the invocation type parameter and set it to be "event".

```
aws lambda invoke  --function-name my-function  --invocation-type
Event  --cli-binary-format raw-in-base64-out --payload '{ "key":
"value" }' response.json
```

Stream (Poll-based) Model

The last model is the stream model, also called the poll-based model. This is typically used when you need to poll messages out of stream or queue-based services such as DynamoDB streams, Kinesis streams and Amazon SQS. The Lambda service runs a poll-er on your behalf, and consumes the messages and data that comes out of them, filtering through them to invoke your Lambda function on only messages that match your use case. With this model, you would need to create an event source mapping to process items from your stream or queue.

An event source mapping links your event source to your Lambda function, so that the events generated from your event source will invoke your Lambda function. These mappings, the response you get back, the permissions you set up, the polling behavior and even the event itself, can be very different based on the event source you're using. However, the way you create event source mappings stays the same. You can do this by using the SDK or CLI. Here is an example of how to create an event source mapping by using the CreateEventSourceMapping CLI command:

```
aws lambda create-event-source-mapping --function-name my-function
--batch-size 500 --maximum-batching-window-in-seconds 5
--starting-position LATEST \
--event-source-arn
arn:aws:dynamodb:us-east-2:123456789012:table/my-table/stream/2019-06
-10T19:26:16.525
```

You can see that in this command, I'm creating a mapping between a DynamoDB stream and my Lambda function. I'm also specifying that the event source mapping batches records together to send to my function. You can control how it batches records using the batch size and the batching window. When your batching size is met or the batching window reaches its maximum value, in this case 5 seconds, your lambda function will be invoked.

Choosing the Right Model

So which one of these models is right for you?

Well, the easiest use case is if you're processing messages from a stream or queue, the best choice for that is to create an event source mapping and use the polling type of invocation for Lambda.

The next case is if your application needs to wait for a response, then synchronous invocation is the best choice and can help you maintain order.

However, if you have a function that runs for long amounts of time, that does not need to wait for a response, then invoking asynchronously is the preferable option, as it offers automatic retries, a built-in queue, and a dead letter queue for failed events.

Now - keep in mind that if an AWS service invokes your function, the ability to select an invocation type is removed. The service gets this choice instead, and selects the invocation method for you. So all of these hard choices go away.

Monitoring your Lambda Function

Fortunately, monitoring and troubleshooting with Lambda is more straightforward than with some of the other AWS compute Services. That's because a lot of important monitoring and logging metrics are already configured with CloudWatch by default and built into the Lambda dashboard.

Let’s take a look at some of the default metrics that are provided to you through the service. I won’t get into every metric that Lambda supports but I will call out some of the main ones. There are three main categories of metrics:

- Invocation metrics
- Performance metrics
- And concurrency metrics

Invocation Metrics

Invocation metrics are related to the outcome of the invocation. For example, there is a metric called invocations that tracks the number of times the function has been invoked. Similarly, in this category, you also have the errors metric which counts the number of failed invocations of the function.

Performance Metrics

Performance metrics on the other hand, follow exactly what the name suggests, and provide performance details about an invocation. These metrics include duration, which measures how long the function runs in milliseconds from when it is invoked until it terminates. This category also supports the IteratorAge metric which is only used for stream-based invocations such as Amazon Kinesis. It measures in time how long Lambda took to receive a batch of records to the time of the last record written to the stream. This IteratorAge is measured in milliseconds.

Concurrency Metrics

Then the last category is concurrency metrics. Before we get into the metrics, let’s understand what concurrency is. When you want to monitor metrics for concurrency, you can use concurrent executions metric, which is a combined metric for all of your Lambda functions that you have running within your AWS account in addition to functions with a custom concurrency limit. It calculates the total sum of concurrent executions at any point in time.

Logs

In addition to these metrics, CloudWatch also gathers log data sent by Lambda to help you better troubleshoot and understand issues. For each function that you have running, CloudWatch will create a different log group.

The log group name will be prefixed with aws and lambda. When you look at Lambda logs, you’ll see details about your function execution. You can also add in custom logging statements to your function that will display in these logs. For example, if you add a print statement to a Python Lambda function, you will see the output of that print statement in these logs. This enables you to push data to CloudWatch logs automatically in addition to the managed messages that are sent by default.

If you want more information - check out the AWS documentation on monitoring with Lambda.

Summary

In this course, I covered a lot of information about AWS Lambda. At this point, you should be familiar with what AWS Lambda is, how it works, and the benefits it can bring to your organization.

If you would like to get some hands-on experience with AWS Lambda, take a look at the following labs titled:

Introduction to AWS Lambda
https://cloudacademy.com/lab/introduction-aws-lambda/

Use AWS Lambda Custom Runtimes to Run Bash Commands
https://cloudacademy.com/lab/use-lambda-custom-runtimes-to-run-bash-commands/

Process Amazon S3 Events with AWS Lambda
https://cloudacademy.com/lab/aws-lambda-s3-events/

Creating Scheduled Tasks with AWS Lambda
https://cloudacademy.com/lab/aws-lambda-scheduled-events/

Once again, my name is Alana Layton and I hope you've enjoyed our time together. If you have any feedback, positive or negative, please contact us at support@cloudacademy.com. Your feedback is greatly appreciated. Thank you and till next time!

Alana Layton, Sr. AWS Content Creator
alana.layton@cloudacademy.com