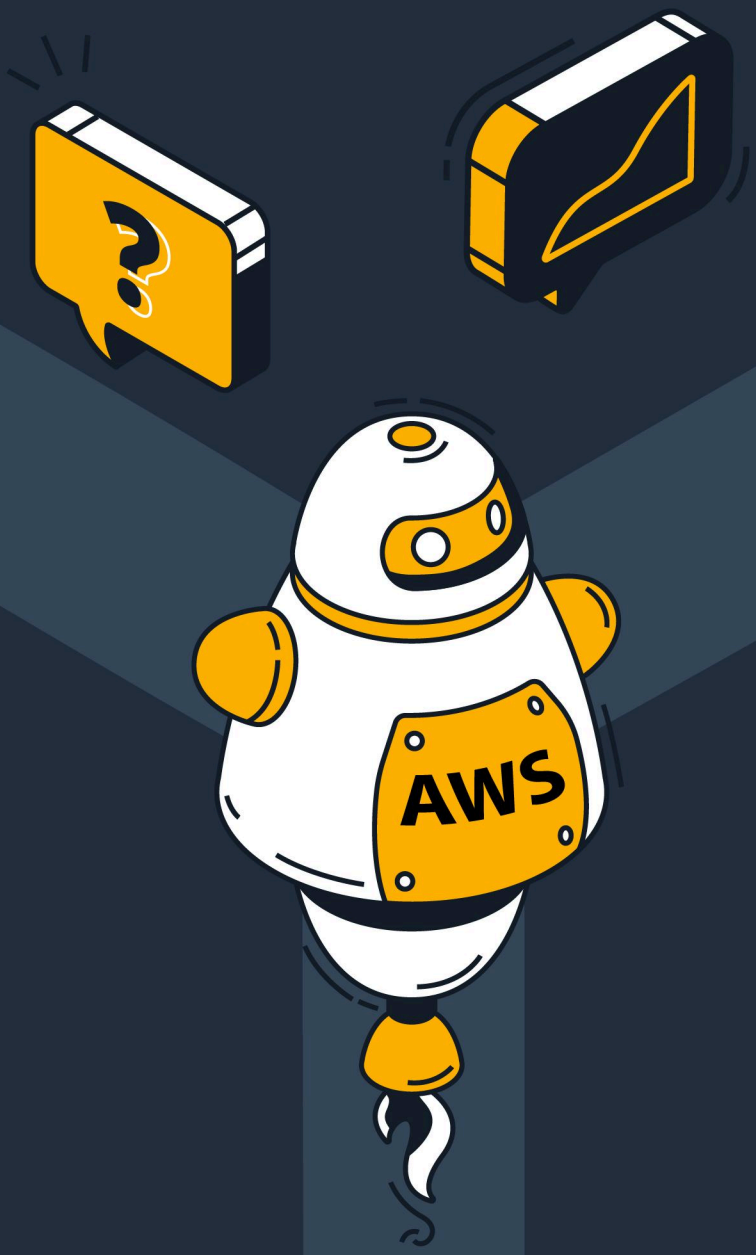
AWS Fundamentals

Tobias Schmidt & Alessandro Volpicella



AWS for the Real World
Not Just for Certifications

AWS Fundamentals

AWS For The Real World

Tobias Schmidt, Alessandro Volpicella

Version v2.1, 2024-10-14

Table of Contents

AWS **Lambda**

5.3. Using Lambda to Run Code without Worrying about Infrastructure

5.3.1. Introduction

Amazon EC2 enables you to stop managing physical servers and focus on virtual machines. With Lambda, launched in 2014, AWS took this one step further by completely removing customers' liabilities for the underlying infrastructure.

The only thing you need to bring is the actual code you want to run, and AWS takes care of provisioning the underlying servers and containers to execute it.

5.3.2. Lambda Abstracts Away Infrastructure Management, but It Doesn't Come without Trade-Offs

If you have never worked with Lambda before, this may be the **most important chapter** as the included information may not seem very intuitive at first.

Let's take a look at how Lambda works under the hood and what trade-offs we have to face due to its on-demand provisioning of infrastructure. Also, let's see what measures we can use to slightly mitigate the limitations we face.

5.3.2.1. Micro-Containers Running Your Code in the Background

One thing that is often missed or misunderstood is that serverless does not mean that there are no servers. They are simply abstracted away and opaque to the application developer.

When an incoming request is made to your Lambda function, AWS will either:

1. **Internally provision a micro-container** and deploy your code into it, or
2. **Reuse an existing container** that has not been de-provisioned yet and is not already busy processing another request.

As you have probably already guessed, the first option comes with a trade-off as resources have to be assigned on demand, which takes a noticeable amount of time.

5.3.2.2. Assigning Resources on-Demand Takes Time - What's Happening in a Cold Start

Let's take a closer look at the first scenario.

It is not surprising that there is a certain amount of bootstrapping time required before our code can be executed. This process of preparing Lambda's environment for code execution is known as a **cold start**.

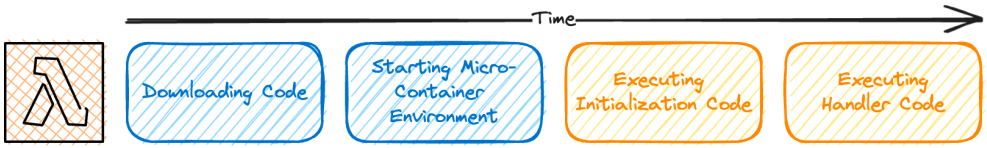


Figure 55. Lambda Cold Start

Lambda needs to download your function's code and start a new micro-container environment which will receive the downloaded code. Afterward, the global code of your function will run. This includes everything **outside** of your handler function. This globally scoped code and its variables will be kept in memory for the time that this micro-container environment is not de-provisioned by AWS. Think of it as a (very) volatile and unpredictable cache.

Lastly, the main code inside your handler function is executed.

Worth noting: Although it's part of the launch phase until your target code finally runs, the global initialization code of your function is not an official part of the cold start.

If we compare this to traditional container approaches, such as running Fargate tasks with ECS, we'll see a difference in the average response times. This is especially noticeable when we focus on the slowest 5% of requests, as they will be much slower in Lambda than on Fargate, as the cold starts will significantly contribute to those.

5.3.2.3. A Micro-Container Can Only Serve One Request at a Time

Each provisioned Lambda micro-container can process only one request at a time. This means that even if there are multiple execution environments already provisioned for a single Lambda, **a cold start may occur** if all of them are currently busy handling requests.

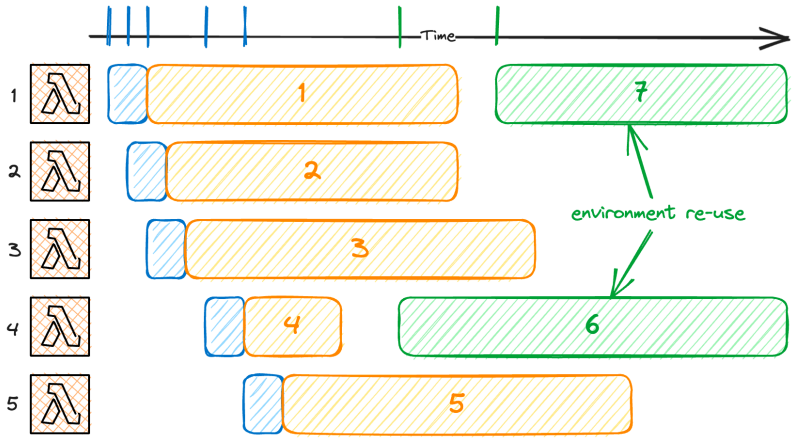


Figure 56. Lambda Micro-Container Reuse

In the invocation scenario described above, it can be observed that five Lambda micro-containers were initially started in the first phase because each consecutive request came in before another container had finished.

The first re-use of a container occurred only at request number six when micro-container number four finished its previous request.

This makes it difficult to reduce cold starts, especially if your application landscape is composed of many different Lambda functions, which may even require mutual synchronous invocations.

5.3.2.4. Global Code Is Kept in Memory and Executed with High Memory and Compute Resources

If we look at a sample handler function, we can see that it's possible to run code outside of the handler method - the so-called **bootstrap code**.

```
bootstrapCoreFramework();
const startTime = new Date();

exports.handler = async (event) => {
    // [...]
    executeWorkload();
}
```

The results of executing this code can create global variables that remain in memory, persisting across multiple executions of the same micro-container. They are only lost after the Lambda environment is torn down.

In our example, the results of our core framework's bootstrap and the start time are kept in memory. They will only disappear when AWS de-provisions our function's container.

This is not the only great thing about the global scope. AWS executes the code outside of the handler method with a high memory configuration (and therefore with high vCPUs), regardless of what you've configured for your function. And even better: **the first 10 seconds of the execution of the globally scoped code are not charged**. This is not a shady trick, but actually a well-known feature of Lambda.

Take advantage of this and bootstrap as much as possible outside the handler function, keeping a global context while your function runs.

A small reminder: Regularly invoke your function via warm-up requests. This will increase the time your global context is kept, as the container lifetime is extended. But it's still limited. AWS will tear down your function's environment after a certain period of time, even if your function is frequently invoked.

5.3.2.5. Your Functions Can Be Invoked Synchronously and Asynchronously

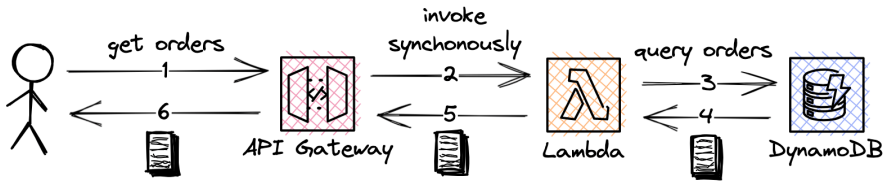


Figure 57. Lambda Synchronous and Asynchronous Invocation

There are two methods to invoke your function's code:

- **Synchronous** or **blocking**: Lambda executes your code but only returns after the execution has finished. You'll receive the actual response that is returned by the function. An example would be a simple HTTP API that is built via API Gateway, Lambda, and DynamoDB. The browser request will hit the API Gateway which will synchronously invoke the Lambda function. The Lambda function will query and return the item from DynamoDB. Only after that, the API Gateway will return the result.
- **asynchronous**: Lambda triggers the execution of your code but immediately returns. You'll receive a message about the successful (or unsuccessful, e.g. due to permission issues) invocation of your function. An example would be a system that generates thumbnails via S3 and Lambda. After a user has uploaded an image to S3, they will immediately receive a success message. S3 will then asynchronously send an event notification to the Lambda function with the metadata of the newly created object. Only then Lambda will take care of the thumbnail generation.

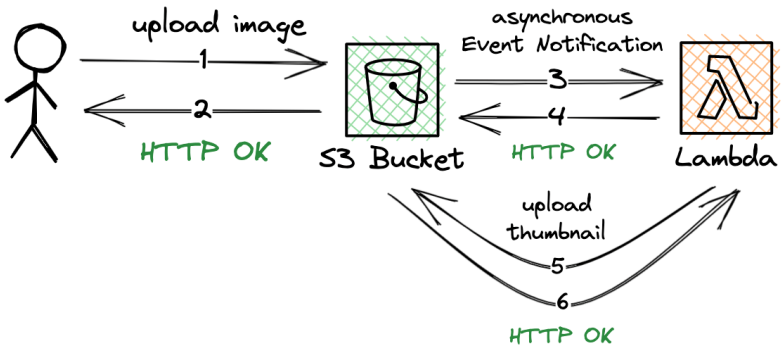


Figure 58. Lambda Synchronous and Asynchronous Invocation

If you're invoking functions from another place, such as another Lambda function, the invocation type depends on how you want to handle results. Synchronous invocation is useful when you need to retrieve the result of the function execution immediately and use it in your application.

```
const AWS = require('aws-sdk');
const lambda = new AWS.Lambda();

exports.handler = async (event) => {
    // returns immediately
    await lambda.invoke({
      FunctionName: 'secondFunction',
      InvocationType: 'Event',
      Payload: JSON.stringify({ message: 'Hello, World!' })
    }).promise();

  // returns after 'myFunction' has finished
    await lambda.invoke({
      FunctionName: 'secondFunction',
      InvocationType: 'RequestResponse',
      Payload: JSON.stringify({ message: 'Hello, World!' })
    }).promise();
```

```
}
```

Let's have a look at the example Lambda function `firstFunction` above. Its sole purpose is the invocation of another function which is called `secondFunction`.

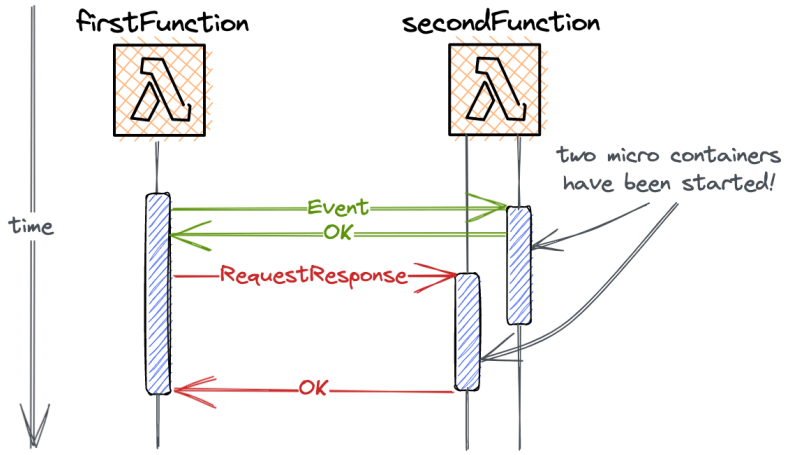


Figure 59. Lambda Synchronous and Asynchronous Invocation

If we look at the sequence diagram above for the invocation of the function `firstFunction`, we can see how both functions execute. The first invocation will return immediately, even though the computation still runs inside the second function. Before this computation can finish, the second invocation occurs and therefore starts another micro-container while the other is still busy. Now, the invocation does not return immediately but waits until the computation has finished.

5.3.3. What's Necessary to Configure to Run Your Lambda Functions

When creating a Lambda function, you need to define many properties of the environment. Some can be changed afterward, some are fixed and can't be changed once the function is created. Let's explore the most important settings and configurations.

5.3.3.1. Choosing the Lambda Runtime and CPU Architecture

There's support for many runtimes at Lambda, including Node.js, Python, Java, Ruby, and Go. Besides deploying your function as a ZIP file, you have the option to provide a Docker container image. It's also possible to bring your own runtime to execute any language by setting the function's runtime to `provided` and either packaging your runtime in your deployment package or putting it into a layer.

You can also configure if you want your functions to be executed by an x86 or ARM/Graviton2 processor. The latter one, introduced in 2021 for AWS Lambda, offers better price performance. Citing the AWS News Blog: "Run Your Functions on Arm and Get Up to 34% Better Price Performance."

All of the environment and CPU architecture settings can't be changed without re-creating your function.

The different runtimes vary in their cold start times. Scripted languages like Python and Node.js do better than Java currently, but the latest release of AWS Lambda SnapStart could change that drastically,

as it will speed up cold starts by an order of magnitude for Java functions.

5.3.3.2. Finding the Perfect Memory Size Which Also Results in a Corresponding Number of vCPUs

The memory size of your Lambda function not only determines the available memory but also the assigned vCPUs. This means that higher settings result in higher computation speeds. You will be billed for GB seconds, so more memory will result in paying more per executed millisecond. However, this does not necessarily mean that your bill will increase linearly with the assigned memory, as more memory and therefore vCPUs will decrease the function's execution time.

5.3.3.3. Timeouts - A Hard Execution Time Limit for Your Function

A single Lambda execution can't run forever.

It's up to you to define a timeout of up to 15 minutes. If an execution hits this limit, it will be forcefully terminated, interrupting whatever workload it is executing right now. The function will return an error to the invoking service if it was synchronously (blocking) invoked.

5.3.3.4. Execution Roles & Permissions - Attaching Permissions to Run Your Functions

Lambda's execution role will determine the permissions it receives on the execution level.

This role will be set when you create your function: either an existing one or a new one. The execution role is important as it determines all permissions that your Lambda function has while it is running. If your function needs to access an Amazon S3 bucket or write logs to Amazon CloudWatch, the execution role must have the appropriate permissions.

As with other services, it is a good security practice to create an execution role with the least privilege. This means it should only have the permissions that are required for the function to perform its intended tasks. This helps to reduce the risk of unintended access to resources and data.

5.3.3.5. Environment Variables For Passing Configurations To Your Functions

Environment variables are key-value pairs that are passed to your Lambda function. Besides your custom variables, you'll find some reserved ones that are available in every function. Those include, among others:

- `AWS_REGION` - the region where your function resides.
- `X_AMZN_TRACE_ID` - the X-Ray tracing header.
- `AWS_LAMBDA_FUNCTION_VERSION` - the version of the function being executed.

As the name already suggests, environment variables are perfect to configure your function for a specific environment. You don't need to hardcode stage-specific variables into the function, but see the function as a blueprint and pass your configuration via the environment.

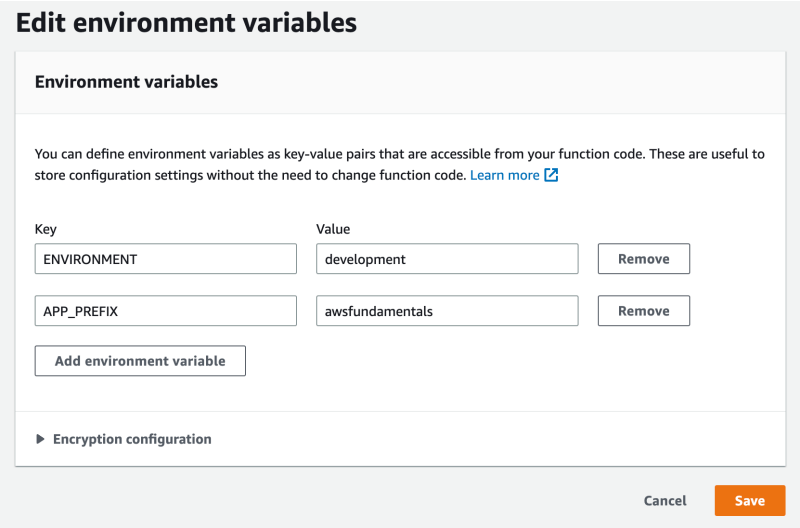


Figure 60. Lambda Environment Variables

Each variable is stored within the function's environment and can be accessed from your code.

- In Node.js, you can use the `process.env` object.
- in Java, you can use the `System.getenv()` method.
- in Python, you can use `os.environ`.

5.3.3.6. VPC Integration - Accessing Protected Resources within VPCs and Controlling Network Activity

There are services that can only be launched inside a VPC, including ElastiCache. If you need to access such a service from Lambda, you'll also need a VPC attachment for Lambda. Other use cases include enhanced security requirements, such as restricting outbound traffic from your functions.

Moreover, running your functions within a VPC will give you greater control over the network environment. If you have functions that do not need internet access, you can put them into a private subnet. This will restrict them from making outgoing calls to the internet, which will immensely increase security.

However, there are considerations when using VPCs. Even though AWS has improved this drastically with the integration of AWS Hyperplane, VPC integration will increase your function's cold start times. Additionally, VPC integration can increase costs, as you'll be charged for data transfer to other resources in your VPC.

5.3.3.7. Natively Invoke Lambda via Different AWS Services via Triggers

Lambda is natively integrated with many other services via triggers, meaning you can launch Lambda functions based on events that are fired from other services.

Prominent examples include:

- Integration with API Gateway to respond to HTTP requests

- Lifecycle events at S3, e.g. launching a Lambda function if an object was created in a specific path of your bucket.
- Consuming events from an SQS queue.
- Scheduling functions based on EventBridge rules.

Triggers are a significant feature for building reliable event-driven architectures that can also recover in case of outages and errors.

5.3.3.8. Triggering Follow-Ups for Successful or Unsuccessful Invocations of Functions via Destinations

The benefit of not having to wait for responses on asynchronous invocation is also the drawback: you can't immediately determine if the execution resulted in any errors. That's why Lambda offers destinations, so you can react to successful or faulty executions.

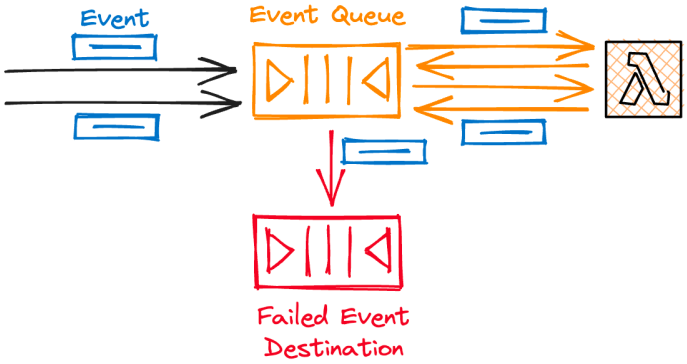


Figure 61. Lambda Destinations

In our example, failed invocations or invocations that cannot be processed are forwarded to an SQS Dead-Letter-Queue. Later on, this queue can be used to investigate events that failed and find out the reason for the failure. Otherwise, we can poll events from the queue from another function to trigger a reprocessing at a later point in time.

5.3.3.9. Code Signing to Ensure the Integrity of Deployment Packages

Your Lambda functions are executed on hardened systems, but how do you ensure that your code was never tampered with? With AWS Signer and code signing, you can create signing profiles to enforce that only code from trusted publishers can be deployed to your functions.

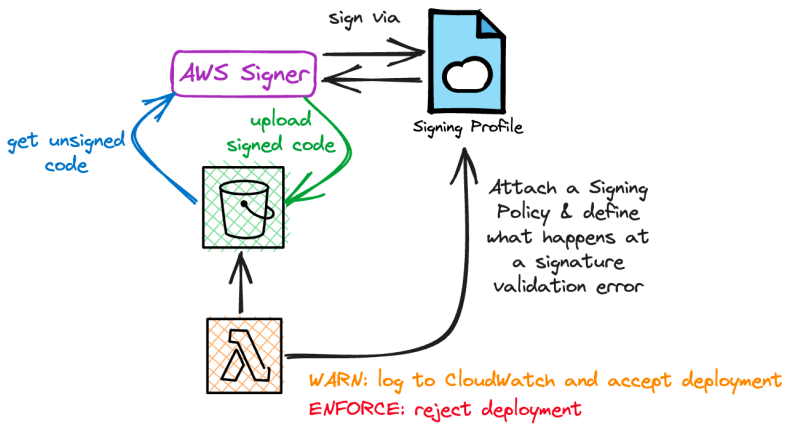


Figure 62. Lambda Code Signing

5.3.3.10. Using Unique Pointers to Functions via Aliases and Versioning

You can have different versions of your function running in parallel. This allows you to test code changes without affecting the currently stable version on your staging environment.

When you publish a new version, you will get a version number that can be used to invoke your function via the qualified ARN:

```
arn:aws:lambda:us-east-1:012345678901:function:myfunction:17
```

In addition, you can create an alias for each version of your function. An alias acts as a pointer to your function.

The benefit of using aliases instead of qualified ARNs is that you can use them with event source mappings without the need to update each mapping after publishing a new version. You only need to update a single resource: the alias itself.

5.3.4. Reserved and Provisioned Concurrency to Guarantee Capacities and Reduce Cold Starts

There are two features that help you manage the performance and scalability of your functions beyond just assigning higher memory settings: **reserved** and **provisioned concurrency**.

Both can improve the performance and scalability of your functions, but they serve different purposes. Reserved concurrency ensures that a certain number of instances of your function are always available to handle requests, while provisioned concurrency keeps instances pre-warmed in anticipation of traffic.

5.3.4.1. Reserved Concurrency for Guaranteeing a Function's Concurrency Capacity

The default concurrent execution limit for a Lambda function in an account is 1000. For new accounts, this limit may be even lower. This means that more than 1000 Lambda functions cannot be executed in parallel. It also implies that a large number of functions can run in parallel, especially if you use fan-out

mechanisms (using a function to trigger other functions which will then trigger more functions) to trigger your Lambda functions.

To restrict the maximum number of parallel executions, use reserved concurrency for your function. Reserved concurrency is subtracted from your account limits so that AWS can guarantee that this scale of parallel executions is always possible for these specific functions. It also ensures that there's never a chance to run more than that number in parallel.

If a function doesn't declare a value for reserved concurrency, it uses the unreserved concurrency capacity left in your account, which could be completely consumed under certain circumstances.

5.3.4.2. Provisioned Concurrency for Reducing Cold Starts

Regardless of the strategies you're using to keep Lambda functions warm via strategic health checks or your level of permanent requests per second, your micro-containers **will be de-provisioned** at some point.

To overcome this, use provisioned capacity. AWS keeps a certain number of Lambda environments provisioned so they're always ready for execution for incoming requests.

This comes with significantly higher pricing and also increased times for deployments (up from a matter of seconds to a few minutes).

5.3.5. Layers Enable You to Externalize Your Dependencies

When building extensive business logic, using existing libraries can often be more efficient than reinventing the wheel. This sometimes results in having more code in dependencies than in actual self-implemented business logic, which can slow down packaging and deployments.

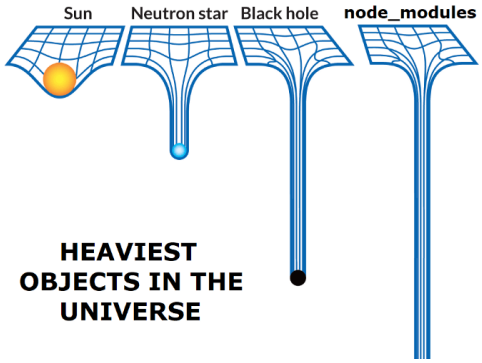


Figure 63. Lambda Layers

In order to include dependencies in your Lambda function, you must package them individually for each function, even if multiple functions rely on the same packages.

Lambda Layers provide a solution to this problem. You can create a versioned Layer that includes the dependencies needed for your Lambda function and attach one or several functions to the same layer. This way, all functions will have access to the included dependencies.

When deploying new functions, you only need to package your own code, which drastically reduces packaging and deployment times, as your code likely only takes up a few kilobytes.

Figure 64. Lambda Layer Size Limitation

There is a size limitation for deployment packages, which includes the size of referenced layers. The limit is 50 MB for zipped files and direct upload, and 250 MB for the unzipped archive.

Make sure to package your dependencies in the correct folder. For instance, Lambda expects your `node_modules` to be located inside the top-level folder `nodejs`.

5.3.6. Monitoring Your Functions with CloudWatch to Detect Issues

Like with other services, Lambda integrates with CloudWatch by default and sends a lot of useful metrics without needing further configurations. CloudWatch also automatically creates monitoring graphs for any of these metrics to visualize your usage.

The default metrics include:

- **Invocations** - The number of times the function was invoked.
- **Duration** - The average, minimum, and maximum amount of time your function code spends processing an event.
- **Error count and success rate (%)** - The number of errors and the percentage of invocations that were completed without errors.
- **Throttles** - The number of times an invocation failed due to concurrency limits.
- **IteratorAge** - For stream event sources, the age of the last item in the batch when Lambda received it and invoked the function.
- **Async delivery failures** - The number of errors that occurred when Lambda attempted to write to a destination or dead-letter queue.
- **Concurrent executions** - The number of function instances that are processing events.

Any log messages you write to the console can also be sent to and ingested by CloudWatch if your Lambda's execution role has sufficient permissions.

- `logs:CreateLogGroup`
- `logs:CreateLogStream`
- `logs:PutLogEvents`

The first permission is only necessary if you don't create the log group yourself. If you create one yourself, you can easily define a retention policy so that log messages expire after a defined period of

time. This helps to avoid unnecessary costs for logs that are no longer in use.

5.3.7. Going into Practice - Creating Our First Serverless Project

We have covered the most important fundamentals of Lambda. Now, let's create our first small Lambda project.

We will divide this into four major steps:

1. **Creating a simple Node.js function.** We will create a small function, adapt and deploy code changes within the AWS management console, and test our function via our own test events.
2. **Adding external dependencies.** We will add Axios as a dependency so we can execute HTTP calls in a more convenient way.
3. **Externalizing dependencies into a Lambda Layer.** Dependencies update rather rarely compared to our own code. Let's extract our new dependency into a Lambda Layer so we don't need to package and deploy them for each code update.
4. **Invoking another Lambda function.** Let's create another function that we can invoke from our initial function to see the differences between synchronous and asynchronous invocations.

5.3.7.1. Creating a Simple Node.js Function

To create a simple Node.js function, go to the AWS Lambda console and click on `Create function`. Define a function name, select your target architecture, and choose which runtime you want to use. For this example, we will use Node.js.

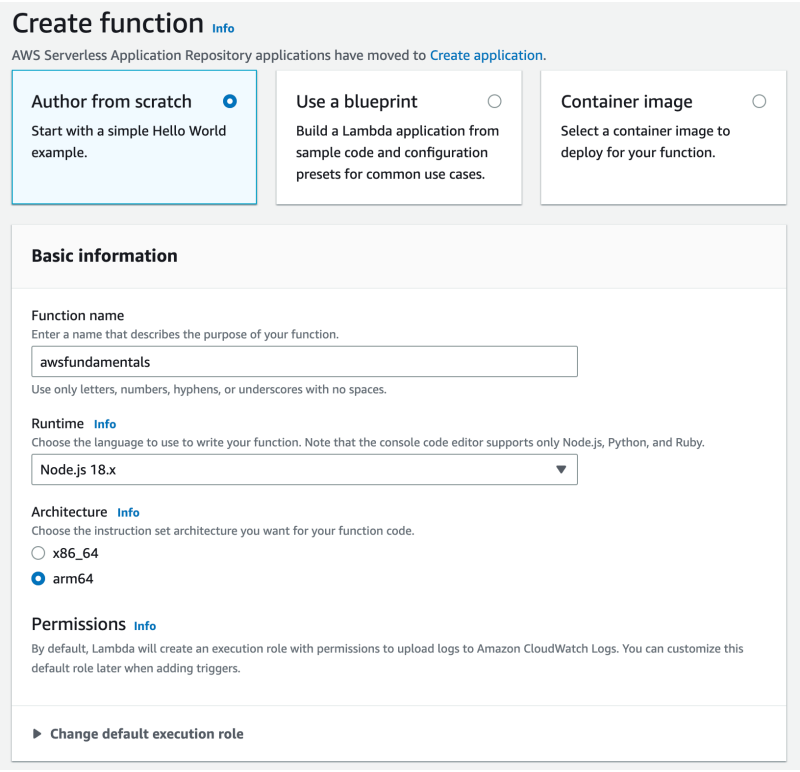


Figure 65. Lambda Function Creation

After clicking `create`, you'll be taken to your functions overview. You'll immediately notice the live editor for our function's code. This editor can also be used to test our function, as well as to save and deploy updates to our function.

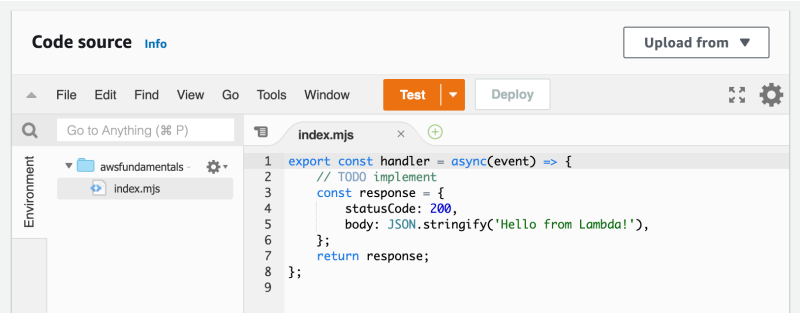


Figure 66. Lambda Code Editor

Let's do exactly that by clicking on `Test`. This will open a modal where we can create our first test event. Let's pass a JSON object with a field `message` to our function.

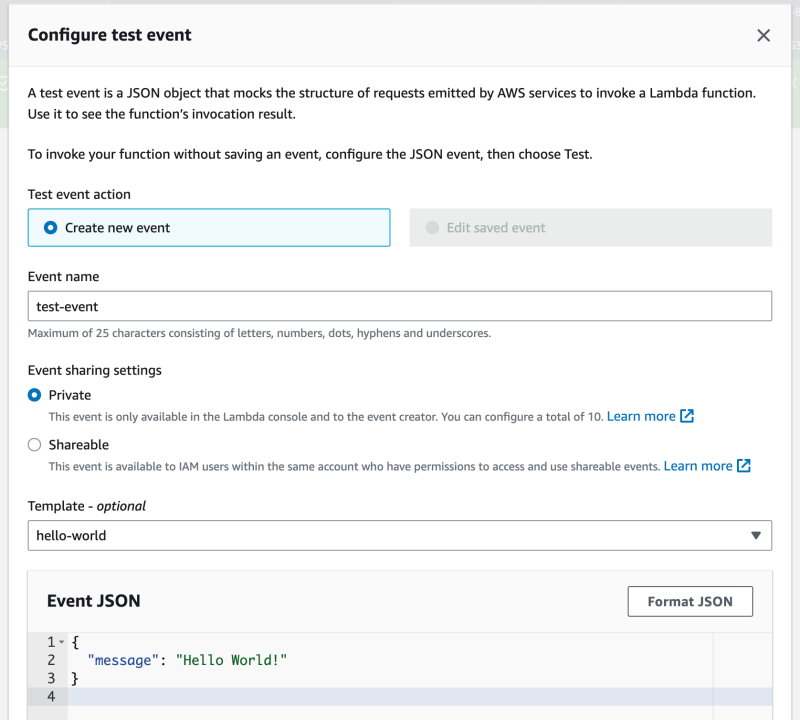


Figure 67. Lambda Test Events

Let's modify our function to return the message passed in the function's response. After clicking `Deploy`, the changes will be deployed to our function. Then, we can invoke our function with the test event.

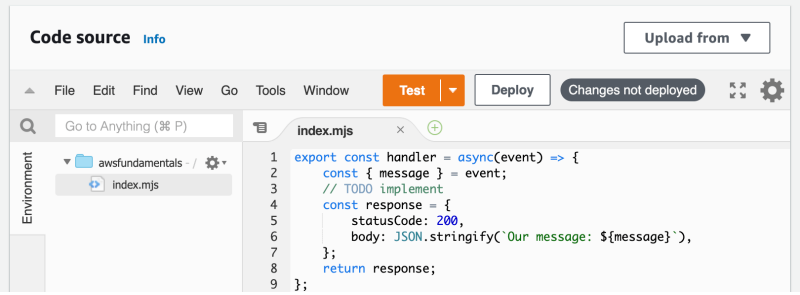


Figure 68. Lambda Function Adaptation

We'll get what we expect: our message!

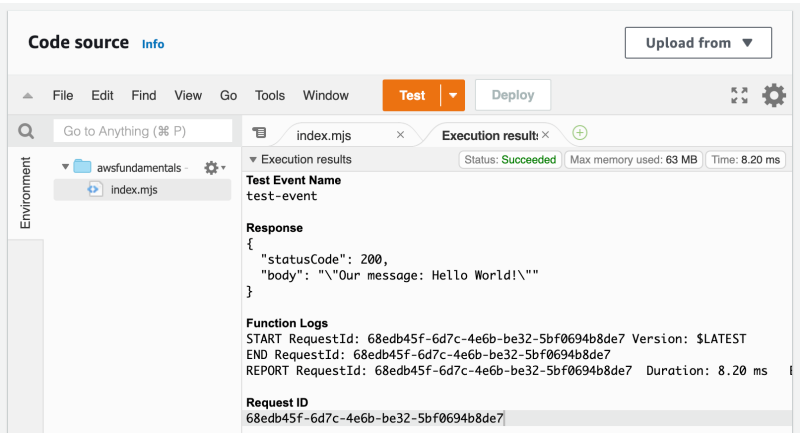


Figure 69. Lambda Test Changes

That's it for the first simple task - you have created, run, updated, and successfully invoked a Lambda function.

5.3.7.2. Adding External Dependencies

Now, let's continue and add some external dependencies. To do this, we will initialize a new project using `npm` and install Axios (a powerful HTTP client library). We will also create a file for our function's code.

```
mkdir awsfundamentals && cd awsfundamentals
npm init && npm i axios
touch index.js
```

Let's rewrite our existing code by adding a new HTTP call via Axios to `https://ipinfo.io/ip` in order to obtain the function's external IP address.

```
const { create } = require("axios");

exports.handler = async () => {
  // create a new axios instance
  const instance = create({
```

```
    baseURL: "https://ipinfo.io/ip",
  });
  // make a GET request to retrieve our IP
  const { data: ipAddress } = await instance.get();
  return {
    statusCode: 200,
    body: `The Lambda function's IP is '${ipAddress}'`,
  };
};
```

Now, we only need to bundle our code together with the `node_modules` folder into a ZIP archive. Afterward, we can upload the archive to our Lambda function via `Upload from > .zip file`.

```
zip dist.zip .
```

After uploading, you can execute it again by clicking on the test button. The response should resemble the following:

```
Response
{
  "statusCode": 200,
  "body": "The Lambda function's IP is '54.77.191.130'"
}

Function Logs
START RequestId: f5e50828-82c2-49a0-96a8-09d343aae66a Version: $LATEST
END RequestId: f5e50828-82c2-49a0-96a8-09d343aae66a
REPORT RequestId: f5e50828-82c2-49a0-96a8-09d343aae66a  Duration: 572.21 ms
Billed Duration: 573 ms Memory Size: 128 MB Max Memory Used: 74 MB  Init
Duration: 266.41 ms

Request ID
f5e50828-82c2-49a0-96a8-09d343aae66a
```

Although our function is currently small because it only includes Axios, its size can quickly increase. Once it reaches a certain threshold, the code becomes impossible to view or edit in the Lambda console. Additionally, we upload many kilobytes for every function upload, even though there may only be changes to our code.

To solve this issue, we will extract our dependencies into a Lambda Layer in the next part.

5.3.7.3. Externalizing Dependencies into a Layer

Creating a Lambda Layer has two significant benefits:

- The size of the deployment unit required for function updates is reduced.
- A layer can be shared with multiple Lambda functions that share the same dependencies.

The process is quick and simple. We only need to include our dependencies in our layer's zip file in an expected directory format. In the case of Node.js, the `node_modules` folder must reside in the root folder, `nodejs`.

```
mkdir -p nodejs
cp -r node_modules nodejs
zip layer.zip nodejs
```

Let's go back to the Lambda console and click on `Layers > Create layer`.

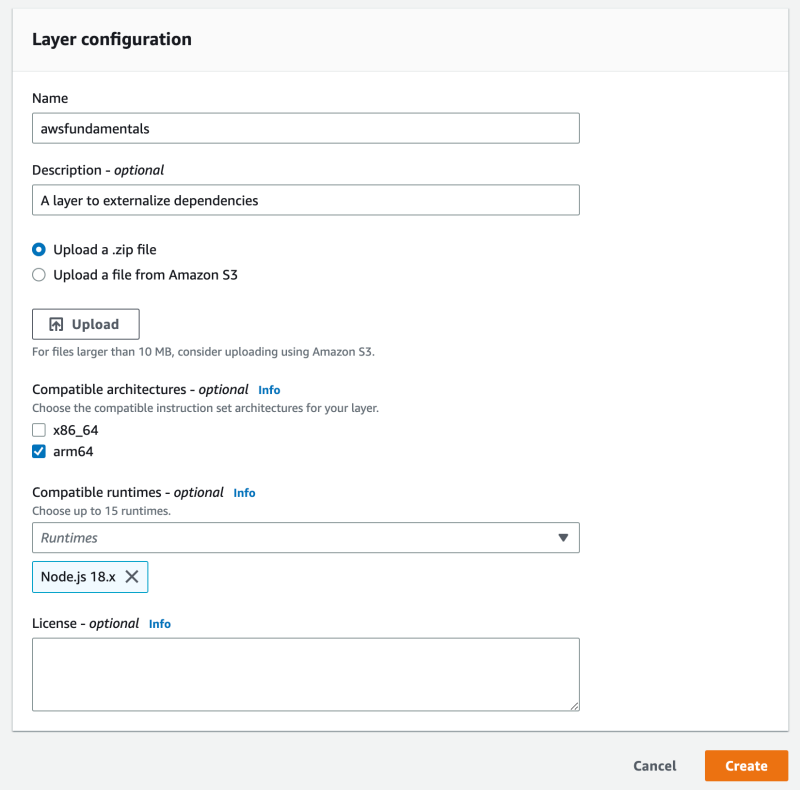


Figure 70. Lambda Layer Creation

After clicking the `Upload` button, select the `layers.zip` file that we created, and finally create the layer by clicking `Create`. Our Lambda Layer will then be ready to use.

Let's go back to our function and scroll down to the layers overview to connect our new layer. Click `Add a layer` to do so.

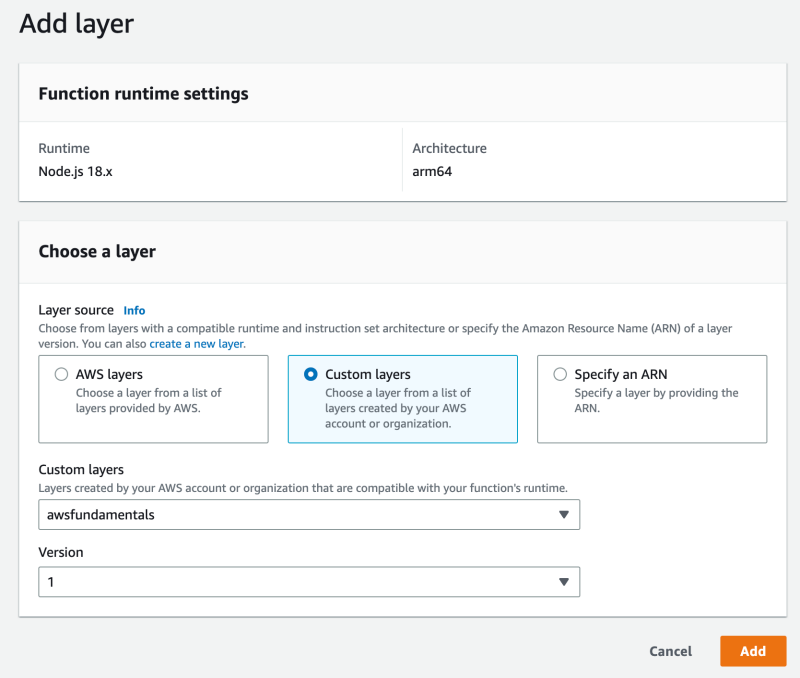


Figure 71. Lambda Layer Attachment

Select custom layers and choose our previously created layer. After clicking `Add`, it will take a few seconds to update the function. Afterwards, you'll be taken back to your function's overview and see that the layer is attached successfully.

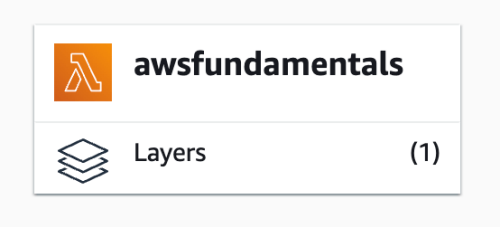


Figure 72. Lambda Layer Attachment

Let's go into the editor of the function and delete the `node_modules` folder by right-clicking and selecting `delete`, as we don't need it anymore. It will be provided via our Lambda Layer.

Let's run our function again to ensure that it still works:

Figure 73. Lambda Layer Test

The third achievement has been unlocked. Let's take on the final step in this small getting-started journey of AWS Lambda.

5.3.7.4. Invoking Another Lambda Function

For the last part, we'll create a second Lambda function that we'll invoke from our initial function. So, head back to the functions overview and click on `Create function`.

To invoke our function, we need two things:

1. Our first function must have the `lambda:InvokeFunction` permission.
2. The AWS SDK.

For the first point, go to the configuration tab of our first function and click on `Permissions`. You'll see the linked execution role. By clicking on the link, you'll be taken to IAM where we can edit the policy.

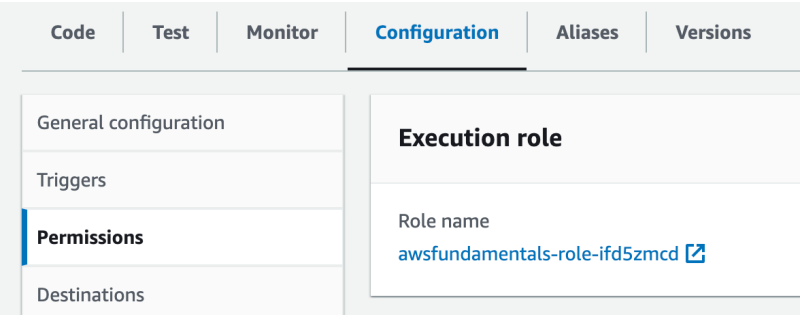


Figure 74. Lambda Invoke Permission

Let's add another permission for Lambda via the visual editor for `InvokeFunction`. Let's only choose our new function here.

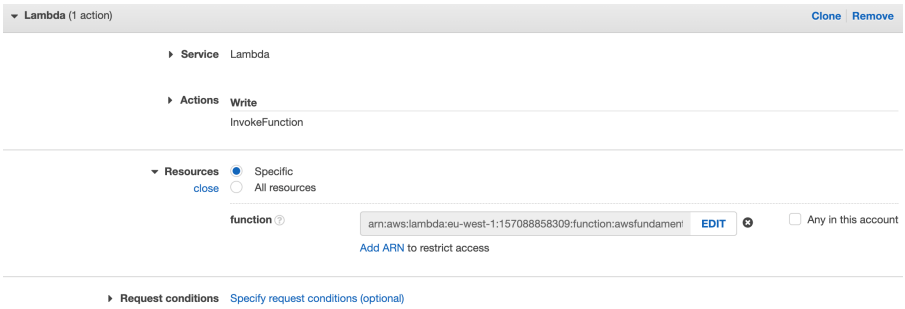


Figure 75. Lambda Invoke Permission

The final JSON policy should now include our new action.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": ["lambda:InvokeFunction"],
      "Resource": [
        "arn:aws:lambda:eu-west-1:157088858309:function:awsfundamentals-invoke"
```

```
      ]
    }
  ]
}
```

Now, we want to update our Lambda Layer to include the AWS-SDK. We'll only do this to see how we can update our layer to a new version. Your Lambda environment always comes with the AWS-SDK, so you don't need to provide it unless you want to pin it to a specific legacy version.

```
### remove the first version
rm -rf layer.zip
### install the AWS-SDK
npm i aws-sdk
### put the node_modules into the right folder
cp -r node_modules nodejs
### package it again
zip -r layer.zip nodejs
```

To upload a new version of the layer, go to your layer and click the `Create version` button. Then, return to the initial function and switch to the new version (2).

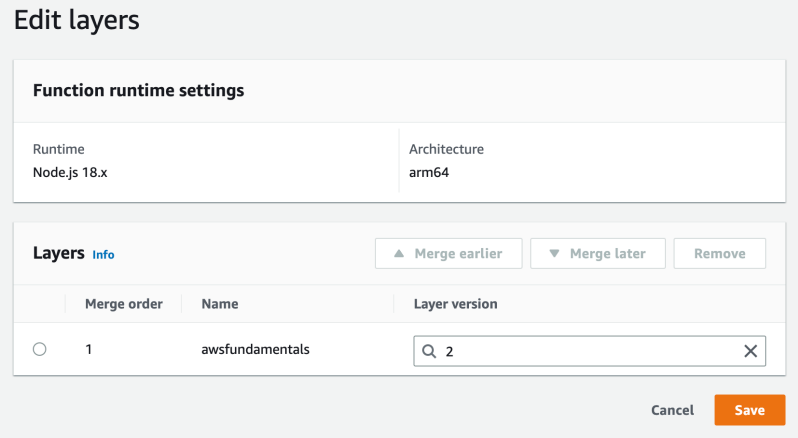


Figure 76. Lambda Layer Version Update

That's done. Let's go back to our second function and adapt our code a bit.

```
export const handler = async (event) => {
  const { waitingPeriodSeconds = 10 } = event;
  await new Promise((resolve) =>
    setTimeout(resolve, waitingPeriodSeconds * 1000)
  );
  return {
    statusCode: 200,
  };
```

```
};
```

We have included some waiting periods before the function returns, so that we can clearly observe the impact of different invocation types. By default, the waiting period is set to 10 seconds, but you can also pass a different value via the incoming event.

Now, let's go back to our initial function and add the invocation part. For the time being, we will invoke the function synchronously.

```
const Lambda = require('aws-sdk/clients/lambda');

exports.handler = async (event) => {
  const startTime = Date.now();
  // Invoke the target function synchronously
  await lambda
    .invoke({
      FunctionName: "awsfundamentals-invoke",
      InvocationType: "RequestResponse",
      Payload: JSON.stringify({
        waitingPeriodSeconds: 5,
      }),
    })
    .promise();
  const endTime = Date.now();
  const elapsedTime = endTime - startTime;
  return {
    statusCode: 200,
    body: `Invocation took ${elapsedTime}ms`,
  };
};
```

Deploy the update and invoke our function.

```
Response
{
  "errorMessage": "Task timed out after 3.01 seconds"
}

Function Logs
START RequestId: 24b0dca3-52b0-4026-afa1-73a143df8e09 Version: $LATEST
2022-12-23T07:57:26.010Z 24b0dca3-52b0-4026-afa1-73a143df8e09
Task timed out after 3.01 seconds
```

Well, that's not what we expected. However, upon examining our function's configuration, it appears

that the default timeout for Lambda is 3 seconds. Since we wait for 5 seconds until our second function finishes its execution, both functions will time out. To resolve this, we need to adjust the timeout setting. Go to `Configuration > General Configuration` and set the timeout to 10 seconds.

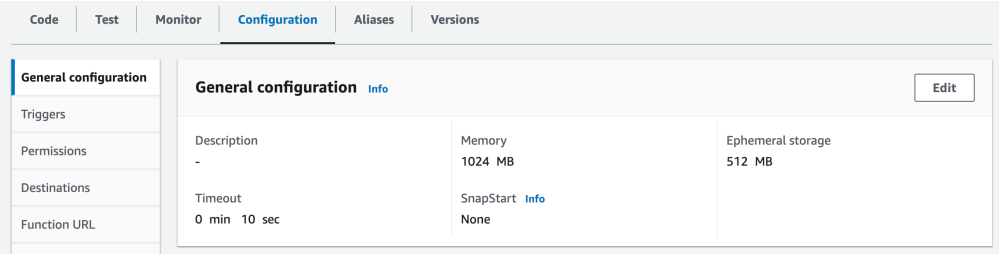


Figure 77. Lambda Timeout Configuration

Afterward, the function is updated, let's retry the invocation.

```
Response
{
  "statusCode": 200,
  "body": "Invocation took 5107ms"
}
```

That's what we expected. Let's switch to the asynchronous function invocation by changing `RequestResponse` to `Event`.

```
const Lambda = require('aws-sdk/clients/lambda');

exports.handler = async (event) => {
  const startTime = Date.now();
  // Invoke the target function synchronously
  await lambda
    .invoke({
      FunctionName: "awsfundamentals-invoke",
      InvocationType: "Event",
      Payload: JSON.stringify({
        waitingPeriodSeconds: 5,
      }),
    })
    .promise();
  const endTime = Date.now();
  const elapsedTime = endTime - startTime;
  return {
    statusCode: 200,
    body: `Invocation took ${elapsedTime}ms`,
  };
};
```

After saving our update and redeploying the function, we will see in the next invocation that the execution time has significantly decreased.

```
Response
{
  "statusCode": 200,
  "body": "Invocation took 719ms"
}
```

Now, our initial function doesn't wait for the second one to finish executing. This results in faster execution, but we can't be sure if the second function finished executing without errors.

We've covered a lot in this small project, which is a great starting point for further experimentation with Lambda.

5.3.8. Exposing Your Function to the Internet with Function URLs

The default way of exposing your Lambda function to the internet via HTTP is by creating an API Gateway. Recently, you can also invoke functions directly via Function URLs, which is a convenient way of exposing your function without creating additional infrastructure.

When enabling function URLs, Lambda will automatically create a unique URL endpoint for you, which will be structured like this:

```
https://<url-id>.lambda-url.<region>.on.aws
```

Your function will be protected via AWS IAM by default, but you can configure the authentication type to `NONE` to allow public, unauthenticated invocations.

CloudWatch also collects function URL metrics such as the request count and the number of HTTP 4xx and 5xx response codes.

5.3.9. Attaching Shared Network Storage with EFS

Lambda only comes with ephemeral storage: the temporary directory `/tmp`. Everything stored here will be kept until your function is de-provisioned. Until recently, this was limited to 500MB and only recently made configurable up to 10GB, but with additional costs.

For a durable storage solution, you can either choose Amazon S3 or EFS. The major advantage of EFS is that it's a typical file system storage and not an object storage - so you can use it like any other directory. You'll need to keep in mind that you'll pay for the EFS storage, the data transferred between Lambda and EFS, and the throughput. You also have to attach your Lambda function to a VPC, as EFS also has a strict VPC requirement. This will increase cold start times.

5.3.10. Running Code as Close as Possible to Clients with Lambda@Edge

With CloudFront, you are able to execute your Lambda functions on the edge. The capabilities are reduced in comparison to traditional Lambda functions, but they still give you a lot of opportunities. We will get into detail about this in the upcoming chapter about CloudFront - we just wanted to include this here for the sake of completeness.

5.3.11. Lambda Is Charged Based on Memory Settings, Execution Times, and Ephemeral Storage

One of the major differences between a virtual machine and a container-based solution is a new model of pricing: you are only paying for the actual time at which your code is executed.

What you will find in the documentation and in the pricing charts is the unit of GB-seconds. This means AWS charges you based on the provisioned memory of your function (the GBs, which also imply the number of vCPUs) and the execution time of your functions.

Let us have a look at the free tier limit of 400,000 GB seconds per month and some different configurations:

- 0.5 GB Memory → 400,000 / 0.5 GB → 800,000 GB-seconds → **9.2 days** of execution
- 1.0 GB Memory → 400,000 / 1.0 GB → 400,000 GB-seconds → **~4.6 days** of execution
- 10 GB Memory → 400,000 / 10 GB → 40,000 GB-seconds → **~0.5 days** of execution

Additional charges apply if you increase the ephemeral storage to over 512 MB.

5.3.12. Recursion Protection to Avoid Exploding Costs on Accidents

A Lambda function has the ability to invoke itself, which can lead to endless recursion if there is no or insufficient cancellation conditions. This can result in unexpected charges and concurrency issues.

To prevent this, Lambda recently introduced a feature to detect and stop recursive loops shortly after they occur. This feature uses AWS X-Ray tracing headers to track the number of times an event has invoked the function. If a function is invoked more than 16 times in the same chain of requests, Lambda stops the next invocation and notifies the user via the health dashboard and email.

Recursive loop detection is enabled by default for all AWS customers and supports certain AWS services such as Amazon SQS and Amazon SNS. The article also provides guidance on how to respond to recursive loop detection notifications and prevent further loops.

5.3.13. Speeding Up Cold Starts with SnapStart

SnapStart for Java on Lambda can enhance the startup performance of latency-sensitive applications by up to ten times at no additional cost, usually without requiring any modifications to your function code. As we've learned, the primary factor contributing to startup latency is the time Lambda takes to initialize

the function. This includes loading the function's code, launching the runtime, and initializing the function code.

SnapStart allows Lambda to initialize your function at the time you **publish a function version**. Lambda captures a snapshot of the memory and disk state of the initialized execution environment, encrypts the snapshot, and stores it for rapid access.

When you first invoke the function version, and as the invocations scale, Lambda launches new execution environments from the stored snapshot instead of starting them from scratch, thereby reducing startup latency.

Currently, this feature is only available for the Java runtime. This makes sense, as Java has one of the slowest cold start times. Lightweight scripting languages like JavaScript do not suffer from this kind of latency during cold starts.

5.3.14. Lambda Comes with Hard and Soft Quotas and Limits

As with every other service, Lambda comes with limitations. Some quotas can be increased via AWS support, but some are fixed and cannot be changed unless AWS updates its policies. Let us have a look at the most important ones as it is likely that you will face them sometime in the future.

- Maximum Concurrency: 1,000 for old accounts; 50 for new accounts
- Storage for uploaded functions: 75 GB
- Function Memory: 128 MB to 10,240 MB
- Function Timeout: 15 minutes
- Function Layers: 5 Layers per Lambda function
- Invocation Payload: 6 MB (synchronous), 256 KB (asynchronous)
- Deployment Package: 50 MB zipped and 250 MB unzipped (this includes the size of all attached layers) & 3 MB for the console editor
- `/tmp` Directory Storage: between 512 MB and 10 GB

AWS is known for regularly updating its quotas so it is worth checking back with the current state at AWS Quotas.

5.3.15. A Deep Dive into Great Lambda Use Cases

Lambda is the most flexible service of all AWS offerings due to its on-demand pricing, low entry barrier, and ease of use. There are almost no limitations to what you can use Lambda for. Additionally, Lambda natively integrates with many other services, often without requiring much or any glue code.

The following list provides just a few examples of the many use cases for which Lambda is known. It is possible to write about simple or extraordinary use cases for weeks or months!

5.3.15.1. Use Case 1: Creating Thumbnails for Images or Videos

Traditional approaches to video or image processing involve uploading files to storage and having a server regularly pull for newly created files. This approach can result in idle server times and unnecessary costs during periods of low traffic.

Figure 78. Lambda Use Case: Thumbnail Generation

Lambda can provide a completely different approach by allowing you to put video conversion or thumbnail generation in its hands, while uploading files to S3. You can attach Lambda invocations to S3 object lifecycle events such as **ObjectCreated**. This means that new files will automatically trigger your function with an event that contains all necessary information about the lifecycle event and file.

With Lambda, you will only be billed for computations, so you won't pay for any idle time.

5.3.15.2. Use Case 2: Creating Backups and Synchronizing Them into Different Accounts

Lambda is the perfect tool for creating backups and synchronizing them between different storages or even accounts to ensure redundancy and high security.

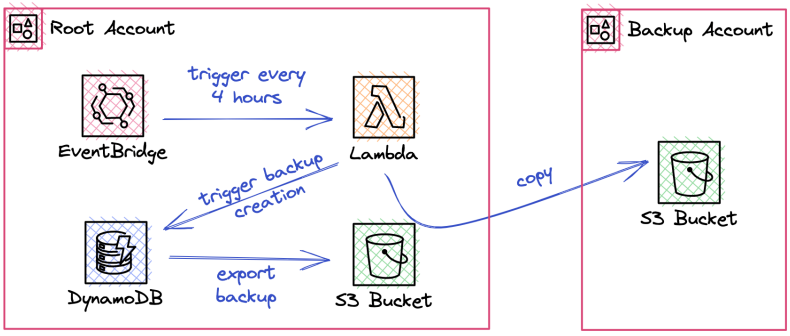


Figure 79. Lambda Use Case: Backup Synchronization

When looking at the previous use case for image and video processing, you can also synchronize files directly based on lifecycle events and event notifications.

5.3.15.3. Use Case 3: Scraping and Crawling Web Pages and APIs and Using the Data for All Kinds of Purposes

You can use Lambda to scrape and crawl web pages or other data sources to gather information for analysis or other purposes.

A common example, with many blog posts written about it, is to automatically update your Twitter banner based on recent followers, recently published blog posts, or any other information you want to display in near real-time.

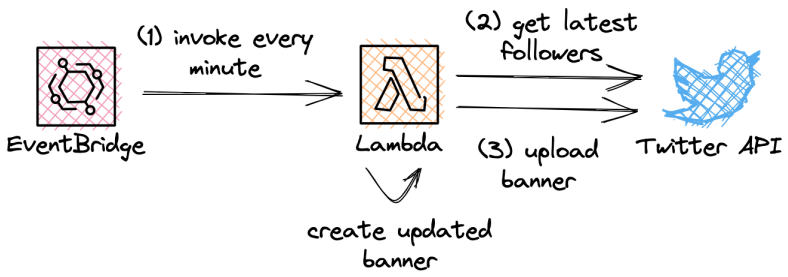


Figure 80. Lambda Use Case: Web Scraping

Lambda can gather the required information from the Twitter API, create a new banner with updated information (e.g. via the sharp library), and then upload the results to your accounts again. The regular invocation of your function is taken over by an EventBridge rule, for example with a schedule for every minute.

5.3.16. Tips and Tricks for the Real World

After hearing about all these great use cases, let's go through a few guidelines on how to make the best out of them. As also mentioned previously, this is not a complete list but a best-practice starting guide.

- **Keep your functions stateless and idempotent**. Function invocations can fail, and retry mechanisms should not result in inconsistent states but always return the same result for the same request. As requests can be executed on different Lambda micro-container environments, it's also important that requests do not need to know a central, in-memory state.
- **Use CloudWatch Alarms.** CloudWatch already collects a bunch of metrics for free, and it's good practice to keep track of them by setting up alarms. It's, for example, important to know when concurrency limits are breached or functions have a high error rate.
- **Pick the right database solution**. Lambda's computing resources are always temporary. Due to this fact, it's difficult to work with database solutions that rely on connection pools, as functions can be de-provisioned at any time, and opening and closing connections all the time is time-consuming. It's practical to use low-latency storages like DynamoDB that can be accessed via the AWS-SDK and do not require connection management.
- Use **Step Functions for orchestrating a large set of functions**. If you need to create multi-step workflows, such as automating a process that involves several Lambda functions and other AWS services, you can use AWS Step Functions.
- **Use Layers for shared dependencies**. If you're working with multiple Lambda functions that require the same external or internal dependencies, outsource them to a Lambda Layer. This will result in less operational overhead.
- **Try to keep your function's code environment independent**. Environment variables are there to store configuration values that are specific to different stages of your application (e.g. development and production). This way, you can easily configure different settings for different environments without hardcoding them into your function's code.
- **Focus on event-driven, resilient architectures**. Invocations of your functions can always fail due to multiple reasons. Timeouts, internal errors, unavailable third parties, and much more. If you focus

on building an event-driven architecture that embraces failure and allows for reprocessing at every step, you'll end up with a resilient system that's able to recover from any failure.

- **Use structured logging**. Rather than using simple text logs, write logs in a structured format like JSON. This makes it easier to search, analyze, and process the logs, as well as to automate certain tasks, such as alerting or aggregating metrics.

5.3.16.1. The Best Practices to Reduce Cold Start Times

Cold starts are a major topic and heavily influence performance and perceived satisfaction with applications. That is why we want to deep-dive into strategies to mitigate or reduce them.

There are many tricks and best practices to reduce cold start times and improve execution times by an order of magnitude:

- **Keep external dependencies minimal**. Think twice if you really need this new dependency and make use of tree-shaking processes (e.g. WebPack for TypeScript/JS) to only include the code in the deployment package that's actually used. Each bootstrap of a micro-container requires your code to be shipped to the container instance and it needs to be loaded when the function starts. Fewer code results in faster launches.
- **Make use of warm-up requests that regularly invoke your functions**. If your function is invoked regularly, the time window until a micro-container is de-provisioned and its resources are free is increased. You can also align the number of parallel warm-up requests to your traffic patterns to start multiple micro-containers in parallel.
- **Bootstrap as much code as possible outside of your handler function**. AWS grants high memory and vCPU settings for code that is executed before your handler function, which means that you'll save additional time until your business code executes. More on this in a later paragraph.
- **Find the sweet spot for your function's memory size**. Less memory and therefore compute resources do not automatically result in a lower bill at the end of the month. Lambda is charged based on the configured memory and the executed milliseconds. Nevertheless, it doesn't mean that at the end of the month, you'll always pay more for a 512MB than for a 2GB function. More vCPUs will result in fewer execution times, especially for computing intense tasks. In other words, it's possible to lower your AWS Lambda bill by increasing memory size. You should monitor your cold starts via CloudWatch and custom metrics, e.g. by writing dedicated log messages and creating custom metrics. Afterward, you can see how different configurations will affect the number of cold starts and their duration.

5.3.17. How to Determine If Lambda and the Serverless Approach Are the Right Fit

As mentioned in the starting chapters, we believe cloud-native is the future. This future heavily revolves around AWS Lambda, as it's the glue that keeps everything together. At the current time, as we've seen with cold starts, there are still some limitations and Lambda is not always the best fit for every requirement.

That's why we want to do a small dive into the requirements analysis to close the Lambda chapter. It's more of an advanced and not a beginner topic, but it's good to have a look into it anyway.

Before you start migrating an existing service or building a new service with a Serverless architecture powered by Lambda, you should ask yourself a set of predefined questions to find out whether Serverless is a fitting approach:

- Does the service need to maintain a **central state**? This can be information that is kept in memory but needs to be shared over all computation resources.
- Does the service need to **serve requests very frequently**?
- Is the architecture rather **monolithic** instead of built out of small, loosely-coupled parts?
- Is it **known how the service needs to scale out** on a daily or weekly basis and how the traffic will grow in the future?
- Are processes mostly revolving around **synchronous operations**?

The more questions answered with **no** the better. If you've answered some questions with yes, it doesn't mean you can't go with Lambda, but you'll face at least some trade-offs in comparison to traditional container technologies like AWS Elastic Container Service (ECS).

5.3.18. Final Words

There's no other service that allows you to quickly build amazing services without spending time on containers, virtual private networks, gateways, and other infrastructure. You've got the code you want to run and Lambda will offer you the environment to do so without having many strings attached. It's also the glue for every Serverless project you'll build, see, or explore in the future.

Personally, we'd also say it's the best service to get started with learning AWS.