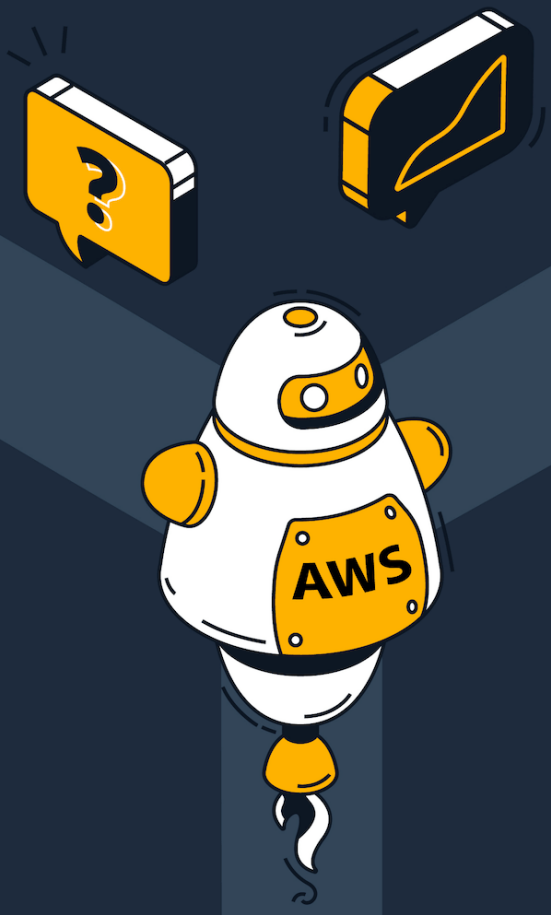
AWS Fundamentals

Tobias Schmidt & Alessandro Volpicella



AWS for the Real World
Not Just for Certifications

AWS Fundamentals

AWS for the Real World - Not Just for Certifications

Preview Edition - Lambda only

Tobias Schmidt

Alessandro Volpicella

Table of Contents

Introduction

With this book, we hope to get you a deeper understanding of AWS, beyond just passing fundamental certifications. It covers a wide range of services, including EC2, S3, RDS, DynamoDB, Lambda, and many more, and provides practical examples and implicit use cases for each one. The book is designed to be a hands-on resource, with step-by-step instructions and detailed explanations to help you understand how to use AWS in real-world scenarios.

Whether you're a developer, system administrator, or even an engineering manager, this book will provide you with the fundamental knowledge you need to successfully build and deploy applications on AWS.

About the Scope of This Book

This book is all about the fundamentals of AWS. The goal is to get you started on how to use AWS in the real world.

First, we'll show you how to create your first AWS Account, how to set up your root users, and how to make sure you will receive billing alerts.

The next part covers the most important AWS services. AWS consists of more than 255 services. We picked out the services that you will use in almost any cloud application. Example services are Elastic Container Service (ECS), Lambda, Simple Queue Service (SQS), Simple Notification Service (SNS), EventBridge, and many more. We dive deep into these services and give you recommendations for the best configuration options, jump into use cases and provide you a list of tips and tricks and things to remember.

In the last part, we give you an introduction to Infrastructure as Code. We want you to understand the differences between various frameworks. For that, we've created a brief introduction and history lesson on IaC. CloudFormation, Serverless, and the Cloud Development Kit (CDK) are three frameworks that are used a lot. We show you examples of how to create infrastructure with all three of them.

Why Did We Bother to Write This?

Why did we bother writing another book about AWS?

Working with AWS both felt like using superpowers. On the one side, you can build applications that are globally available without caring about infrastructure. On the other side having the skill of using AWS is globally in demand.

We want to pass on the knowledge of both points as well. By knowing how to build on AWS you can boost your career. But you can also finally work on your SaaS idea.

We're both lucky in the way we learned AWS. During our studies, we worked in companies where experienced employees could teach us the basics directly but we've also had the freedom to learn ourselves and make mistakes. Through the years, we could harden our skills and gain a lot of insights into different areas. We've seen how AWS progressed, but the fundamentals still remained the same.

We saw colleagues and friends struggling a lot with learning the core services of AWS service and its underlying principles and how to apply them in the day-to-day work. The typical learning path is to get started with certifications. While this is not inherently a bad way it is often not enough. Certificates can be really hard to master. But they often don't bring enough value if you don't put the learnings into immediate practice. People are often still overwhelmed by which services they should use in which situation and how to configure them accordingly. This is the main motivation of this book.

Learning AWS doesn't need to be hard. It is important to focus on the basics and to understand them well. Once this is done all new services or features can be understood really well.

Each cloud application consists of the same set of services and principles.

We both never thought about writing a book. But during our time working, and especially once we started to create content we saw the need. There were so many questions and misconceptions that we wanted to create a resource on how to learn AWS for the real world**.**

Who Is This Book For

This book is for everybody who wants to learn about the fundamentals of AWS. We cover the core building blocks of AWS and Infrastructure as Code.

We will show you example use cases and configuration options for each service. With that, you are ready to understand how to apply it in the real world.

Programming experience doesn't matter for this book. While infrastructure is code nowadays you don't need to know any specific programming language. Programming is a tool you will use to build on the cloud, but it is not a prerequisite as it can be acquired along the path.

This book is also for everybody who did some certifications like the Cloud Practitioner or Solutions Architect Associate but is still overwhelmed with how to apply the learnings in real-world projects.

If you're an entrepreneur who wants to start building on AWS this is also a great resource for you on how to get started.

Or if you are the technical manager of a team and somehow lost contact with AWS and its configuration options you can brush up your knowledge fast and reduce the knowledge gap in your engineering team.

If you are working with AWS for quite some time but still are not sure about some configuration basics (like when to use long & when to use short polling), this is also for you.

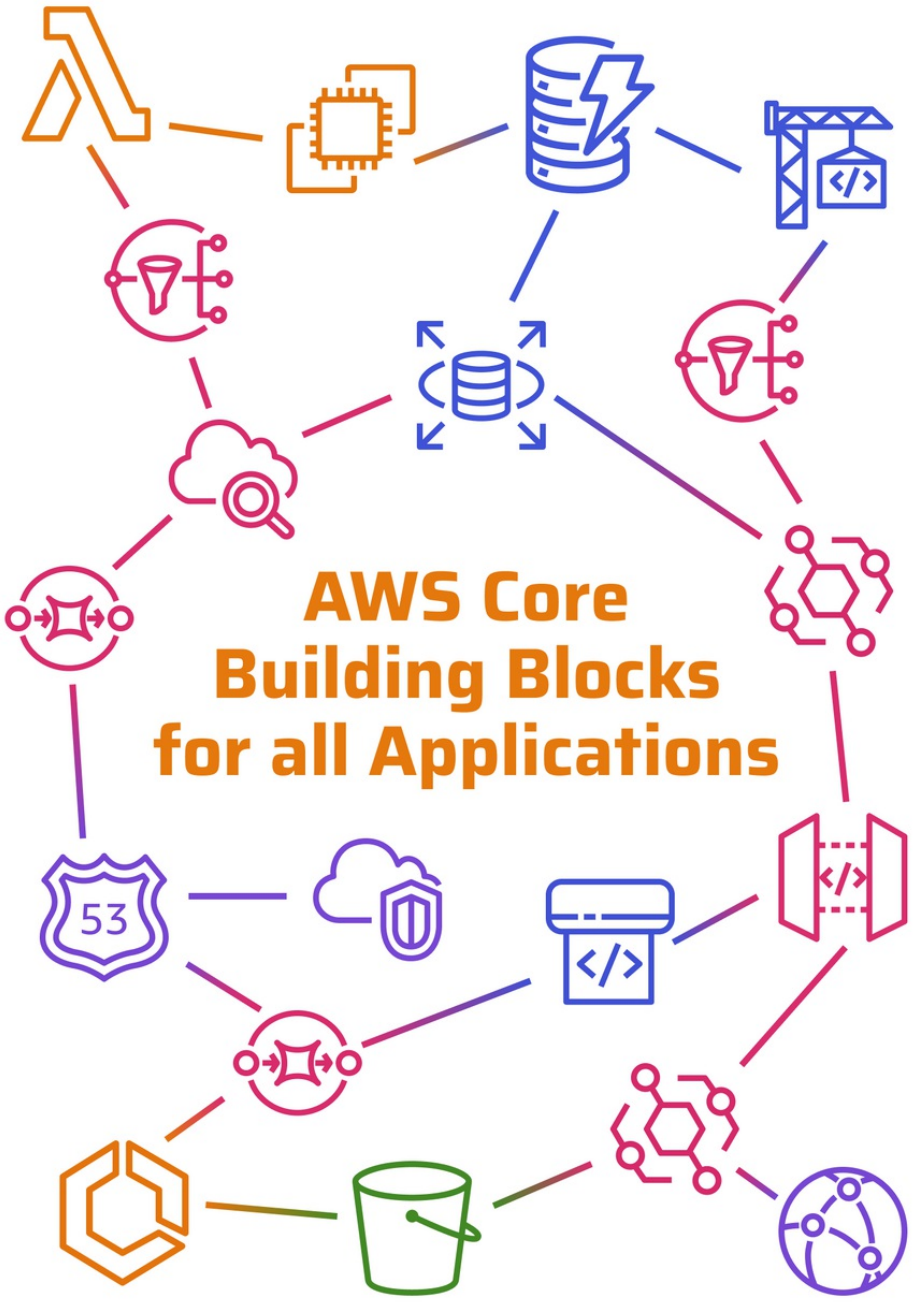
Who Is This Book Not For

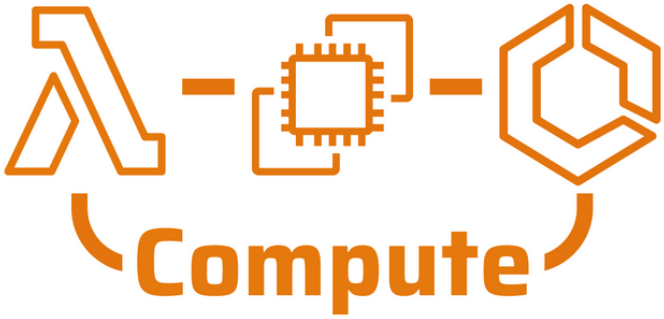
Honesty is important. We only want people to buy this book if they can profit immensely from reading it.

Firstly, if you're really proficient with AWS and you've worked in the area for many years, likely this book is not for you. We don't require previous knowledge about basically anything, and that's also where we start. By exploring every core service as deeply as possible, we want to give aspiring cloud engineers a fundamental tool to start building their own applications or simply to get hired in this area.

This book is also not for people that don't want to or simply won't directly or indirectly work with AWS. If your future or current focus is Azure or Google Cloud Platform, there's more value in purchasing another book. It can make sense to understand how AWS is handling things, but if you aim to work with another cloud provider, learning their specifics is key.

Furthermore, this book doesn't focus on passing certifications. You'll learn the principles that are required to know how to build applications from scratch and how to apply that knowledge, but passing certifications often require very deep knowledge that goes way beyond. This book is a good tool to set yourself up for a good baseline for fundamental certifications like the Cloud Practitioner or the Solutions Architect Associate. But if you focus on passing certifications, doing practice exams, or courses that strictly focus on exam questions, will do a much better job.


AWS Core
Building Blocks
for all Applications
53


Compute


AWS Lambda

Using Lambda to Run Code without Worrying about Infrastructure

Introduction

Amazon EC2 enables you to leave managing physical servers behind and just focus on virtual machines. With Lambda, launched back in 2014, AWS took this one step further by completely removing customers' liabilities for the underlying infrastructure. The only thing you bring is the actual code you want to run and AWS takes care of provisioning the underlying servers and containers to execute it.

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

If you've never worked with Lambda before, this is maybe the most important chapter as the included information doesn't seem to be very intuitive in the first place. Let's have a look at how Lambda works under the hood and which trade-offs we have to face due to its on-demand provisioning of infrastructure. Also, let's see which measures we can use to slightly mitigate the limitations we face.

Micro-Containers in the Back Which Run Your Code

One thing that's often missed or misunderstood is: Serverless doesn't mean that there are no servers. These are just abstracted away and intransparent for the application developer.

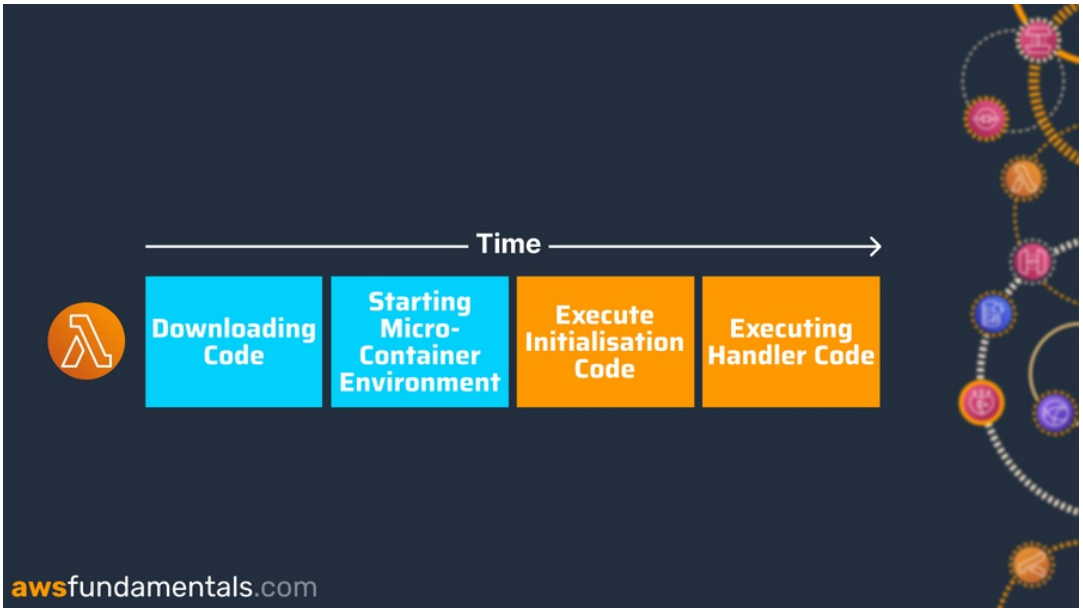
For an incoming request to your Lambda function, AWS will either

1. internally provision a micro-container and deploy your code into it or
2. re-use an existing container that hasn't been de-provisioned yet and is not already busy processing another request.

As you've 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.

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

Let's take a deeper look at the first scenario. We're not surprised that there's a certain amount of bootstrapping time needed until our code is executed. The process of preparing Lambda's environment so it is able to execute your code is called **cold start**.



Lambda needs to download your function's code and start a new micro-container environment that will then receive and execute it. Afterward, the global code of your function will run. This is everything that's 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. See this as some very volatile cache.

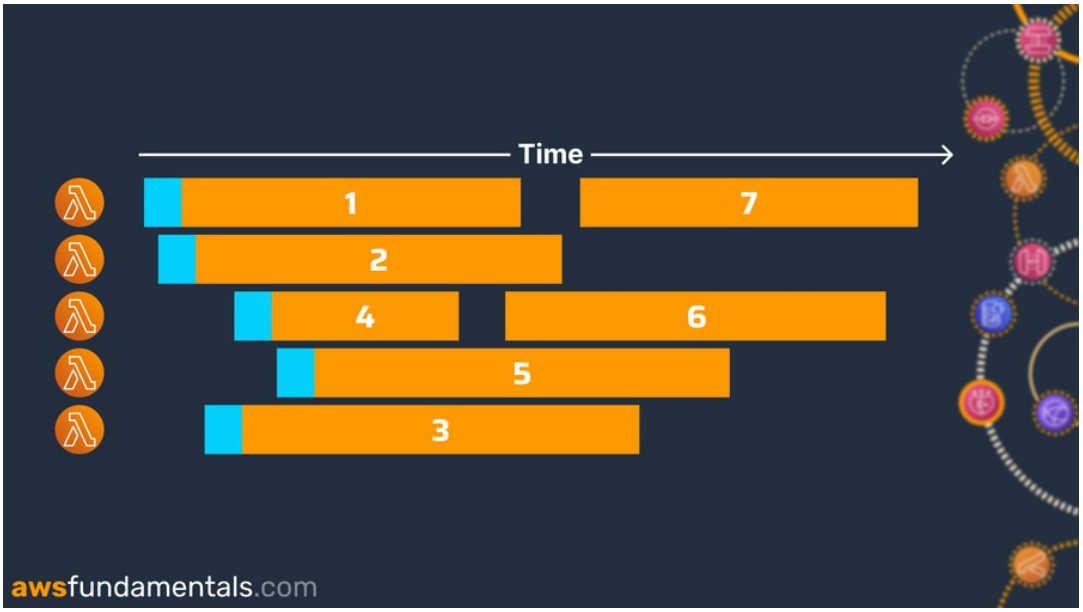
Lastly, your main code is run that's inside your handler function.

Worth noticing: Even though 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, e.g. 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 slower in Lambda than on Fargate, as the cold starts will immensely contribute to those.

A Micro-Container Is Only Able to Serve One Request at a Time

Each provisioned Lambda micro-container is only able to process one request at a time, which means: even if there are multiple execution environments already provisioned for a single Lambda, **there can be another cold start** if all of them are currently busy handling requests.



Looking at the invocation scenario above you can see that five Lambda micro-containers were initially started in the first phase. This was due to the fact, that each consecutive request came in before another container has finished.

The first re-use did only happen at request number six, as micro-container number three (counting top-down) has finished its previous request.

This increases the difficulty of reducing cold starts, especially if your application landscape is built via many different Lambda functions, maybe even requiring mutual synchronous invocations.

Global Code Is Kept in Memory and Is Execute with High Memory and Compute Resources

If we're looking 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 this code execution can result in **global variables that are kept in memory,**

so they continue to exist over **several executions** of this single micro-container. It's only lost after the tear-down of the Lambda environment was executed.

In our example, we'd keep the results of the bootstrap of our core framework and the start time in memory. Those will only vanish when our function's container will be de-provisioned by AWS.

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

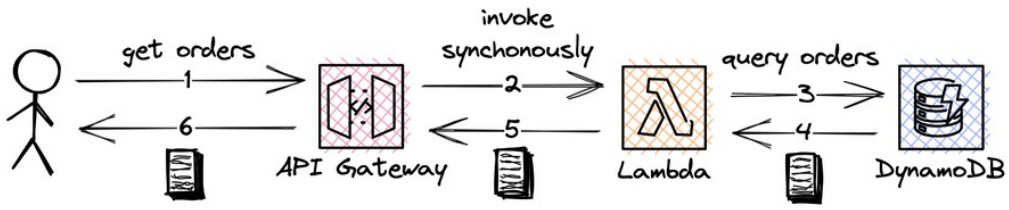
Make use of this and bootstrap as much as possible outside the handler function and keep a global context while your function is running.

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

Your Functions Can Be Invoked Synchronously and Asynchronously

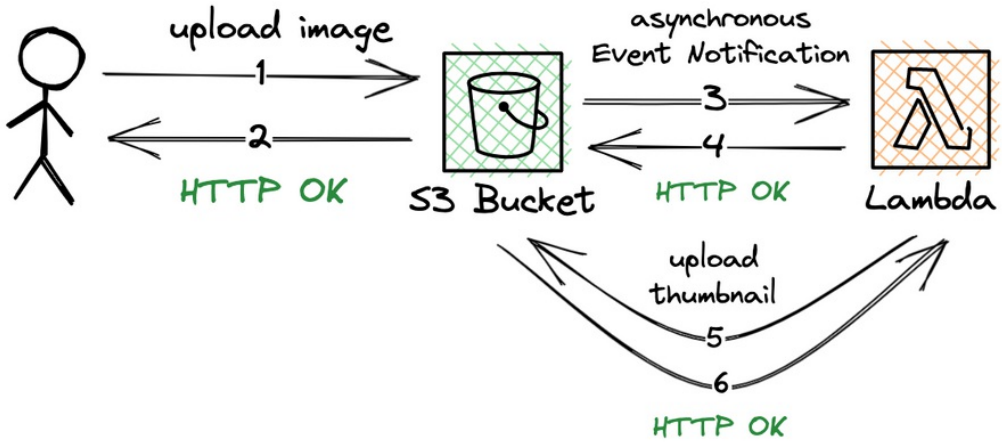
There are two methods to invoke your functions code:

- **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.



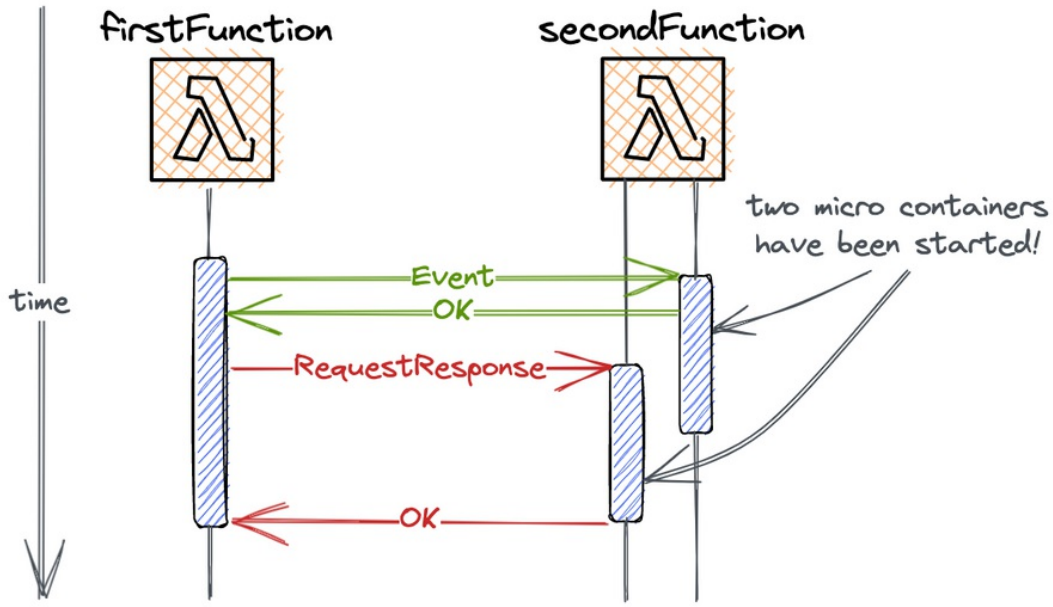
If you're invoking functions from another place, e.g. another Lambda function, the invocation type depends on how you want to handle results. Synchronous invocation is useful when you need to retrieve the result of the function execution immediately and use it in your application.

```
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`.



If we look at the sequence diagram above for the invocation of function `firstFunction`, we see how both 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 hits and therefore starts another micro-container as the other is still busy. Now, the invocation does not return immediately but waits until the computation has finished.

What's Necessary to Configure to Run Your Lambda Functions

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

Choosing the Lambda Runtime and CPU Architecture

There's support for a lot of runtimes at Lambda, including Node.js, Python, Java, Ruby and Go. Besides deploying your function as a ZIP file, you have to option to provide a Docker container image. It's also possible to bring your own runtime to execute any language by setting the functions 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 a 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.

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

The memory size of your Lambda function does not only determines the available memory but **also the assigned vCPUs**, meaning that higher settings result in higher computation speeds. You'll be billed for GB seconds, so more memory will result in paying more per executed millisecond.

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.

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.

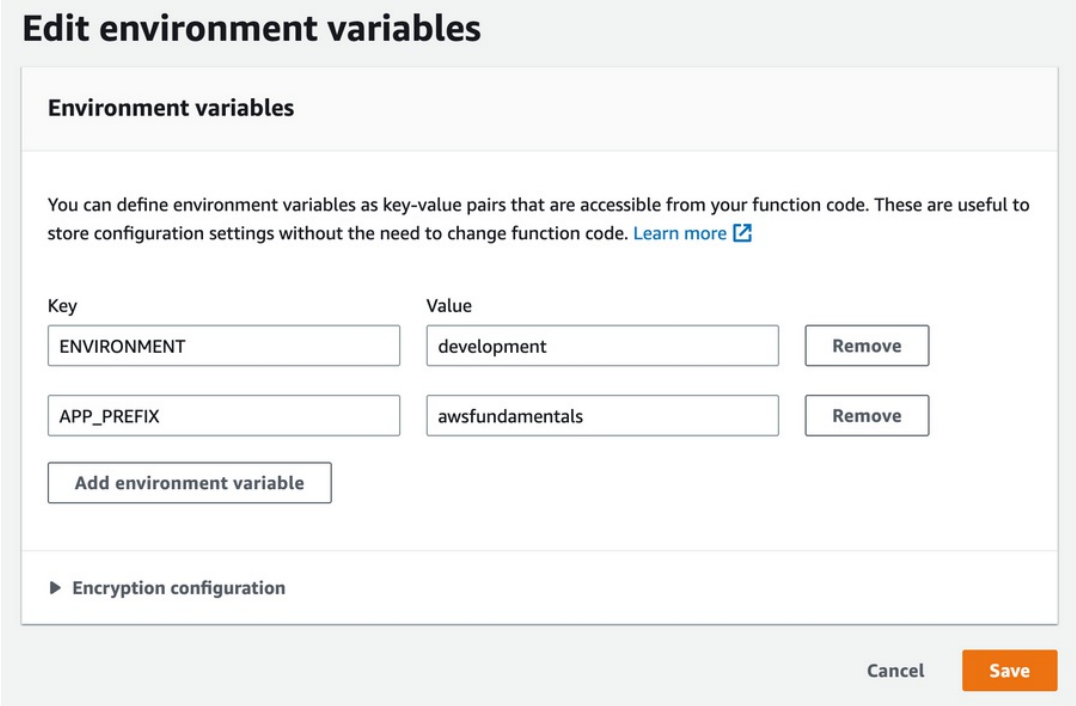
As with other services, it is a good security practice to create an execution role with the **least privilege.** This means it should only have the permissions that are required for the function to perform its intended tasks. This helps to reduce the risk of unintended access to resources and data.

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 which 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.



Each of your variables is stored in the function's environment and can be accessed from your code. For Node.js you can use the `process.env` object, for Java it will be the `System.getenv()` method, and Python will provide `os.environ`.

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 are enhanced security requirements like restricting outbound traffic from your functions.

Also, 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.

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

Natively Invoke Lambda via Different AWS Services via Triggers

Lambda is natively integrated with a lot of other services via triggers, meaning you're able to launch Lambda functions based on events that are fired from other services.

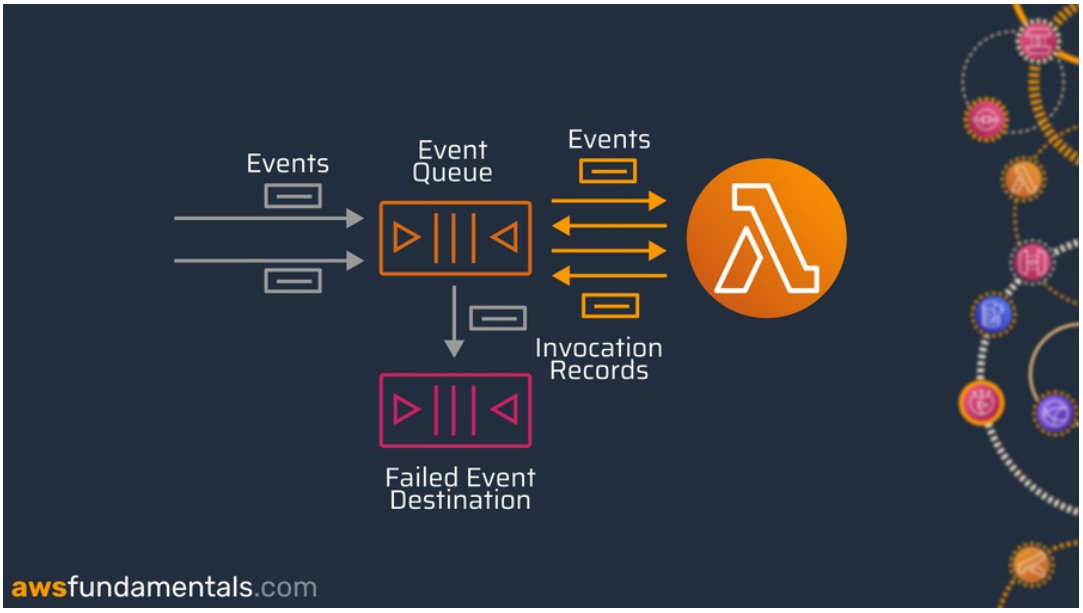
Prominent examples are:

- 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 major feature to build reliable event-driven architectures that are also able to recover in case of outages and errors.

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

The upside of not having to wait for responses on asynchronous invocation is also the downside: you can't immediately decide if the execution didn't result in any errors. That's why Lambda offers destinations so you can react to successful or faulty executions.



In our example, failed invocations or invocations that can't 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.

Code Signing to Ensure the Integrity of Deployment Packages

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

Using Unique Pointers to Functions via Aliases and Versioning

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

When publishing a version you'll get another version number which can be used to invoke your function via the qualified ARN:

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

Additionally, you can create an alias for each of your versions. An alias acts as a pointer to your function. The benefit of using aliases instead of the qualified ARNs is that you can use them with event source mappings without having to adapt each of the mappings after you've published a new version. You'll only need to update a single resource: the alias itself.

Reserved and Provisioned Concurrency to Guarantee Capacities and Reduce Cold Starts

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

Both can help you improve the performance and scalability of your functions, but they are used for different purposes. Reserved concurrency is used to ensure that a certain number of instances of your function are always available to handle requests, while provisioned

concurrency is used to keep instances pre-warmed in anticipation of traffic.

Reserved Concurrency for Guaranteeing a Functions Concurrency Capacity

The default concurrent execution Lambda for an account is 1000. This means it's not possible to have more than 1000 Lambda functions executed in parallel. This also implies that it's possible to run a huge number of functions in parallel, maybe by accident due to recursion with missing exit conditions or similar errors.

For restricting the maximum number of parallel execution you can use reserved concurrency for your function. Each reserved concurrency will be 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 didn't declare a value for reserved concurrency, it will use the unreserved concurrency capacity that is left in your account which could probably be completely consumed under certain circumstances.

Provisioned Concurrency for Reducing Cold Starts

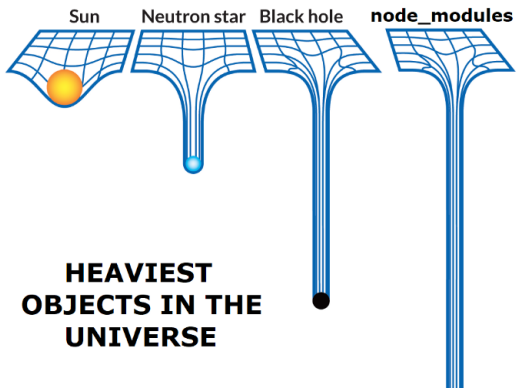
Regardless of the strategies you're using of keeping Lambda functions warm via strategic health checks or your level of permanent requests per second, your micro-containers **will be de-provisioned** at some time.

The only way to get around this is to use provisioned capacity. AWS will keep a certain number of Lambda environments provisioned so they are 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).

Layers Enable You to Externalize Your Dependencies

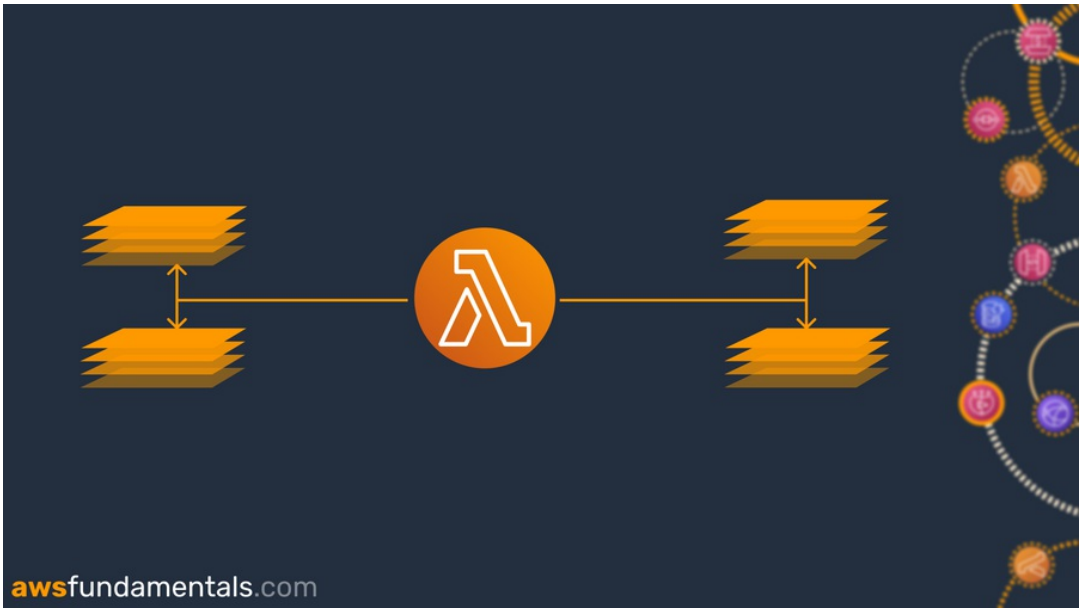
Building extensive business logic often doesn't require you to reinvent the wheel, but to make use of existing libraries. This mostly results in having more code in dependencies than in actual self-implemented business logic which will slow down packaging and deployments.



Also, you'll need to package dependencies for all of your Lambda functions individually as they need to be included in the deployment package - even in the case that most of your function do rely on the same packages.

The solution for this is Lambda Layers. You can create a versioned Layer including the dependencies you need for your Lambda function. Afterward, you can attach one or several functions to the same layer. All of them will get access to the included dependencies.

New function deployments will only require you to package your own code which will drastically increase packaging and deployment times, as you likely only have a few kBytes of code left.



There's a deployment package size limitation, which **includes the size of referenced layers** of 50 MB for zipped files and direct upload and 250 MB for the unzipped archive.

Make sure you package your dependencies in the right folder. For example, lambda expects your `node_modules` to be inside the top-level folder `nodejs`.

Monitoring Your Functions with CloudWatch to Detect Issues

As with other services, Lambda integrates with CloudWatch by default and submits a lot of useful metrics without any 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 that 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 error.
- **Throttles** – The number of times that 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 submitted 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 not in use anymore.

Going into Practice - Creating Our First Serverless Project

We've gone through the most important fundamentals of Lambda. Let's jump into the doing and create our first, own small Lambda project.

We'll divide this into a journey of four major steps:

1. **Creating a simple Node.js function**. We'll create a small function, and adapt and deploy code changes within the AWS management console. We'll also test our function here via our own test events.
2. **Adding external dependencies**. We'll 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.

Creating a Simple Node.js Function

Jump into the AWS Lambda console and click on `Create function`. We only need to define a name, select our target architecture, and chose which runtime we want to use. For our example, we'll go with Node.js.