



# Introduction to AWS Lambda: A Starter Guide

 [blog.awsfundamentals.com/starter-guide-things-i-wish-i-knew-before-for-aws-lambda](https://blog.awsfundamentals.com/starter-guide-things-i-wish-i-knew-before-for-aws-lambda)

Tobias Schmidt



 Play this article

▶ 0:00 / 15:55   

Serverless is becoming a widespread topic and is discussed everywhere. If somebody is talking about serverless, most of the time they are referencing the general idea of leaving the necessity of provisioning and managing servers or containers behind. Known providers like AWS, Azure, or GCP are offering **Functions as a Service** (FaaS) which realizes this idea. Functions are fine-grained units that can be executed directly without provisioning or managing any underlying infrastructure.

There are a lot of benefits, but also trade-offs that should be considered before building your architecture solely around Lambda.

This article provides you with some guidance & hopefully reduces the chances of going wrong at some crossroads for your next project.

## Benefits and Trade-Offs with AWS Lambda

---

Unsurprisingly, there's a lot on both sides.

### Arguments for Going with Lambda

---

- **Simplicity regarding operations** - you don't need to set up anything besides your packaged code. Also, you don't have to worry about failing containers, security patches, or anything else you have to invest your time in if you run your servers or containers.
- **Scalability out-of-the-box** - if you're running a microservice with a framework that offers you fast start-up times, you can scale out (and in) horizontally almost instantly based on the current demand.
- **Pay-as-you-go** - you're paying exactly (at 1ms granularity!) for what you're using. If your function isn't executed, there are zero costs.
- **Agility and development speed** - reduces a lot of burdens for developers so they can focus more on application design and actual coding. Also, allows drastically fast development of experiments, spikes, and Proofs-of-Concept.

### Constraints and Limitations

---

- **Cold starts** - If you do not want to use provisioned concurrency, your function **will** be de-provisioned at some time, causing some time to come back up at the next invocation. It's possible to do tricks to keep customer-facing cold starts low, but it can't be completely avoided.
- **Possible traps with different languages** - if you're not too familiar with the technology, you can encounter unexpected behavior, for example, that your node function freezes while there are events left in your event loop because it reached the end of the execution stack.
- **Application affinity** - Migrating existing services containing a lot of business logic can be especially difficult because most likely there are architectural adaptations necessary to take advantage of Lambda's benefits.
- **High abstraction goes with lower predictability** - Amazon does not provide too many details about how the underlying magic is working. Also, they are not giving any guarantees about how long your function's instance will be kept in memory or how it's internally decided on which function instance gets invoked.

- **Pricing** - if you want to go full serverless, you're also forced to use API Gateway which does come with additional costs if you're expecting a lot of requests. Depending on the Lambda memory size you assign, processing time can also be a lot higher in comparison to a process running in a container in Fargate.
- **Vendor-lock** - contrary to applications running in containers, it's much more difficult to move providers, because you'll likely take advantage of additional features provided by AWS.

## The Importance of Requirements Analysis

---

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

1. Does the service need to maintain a central state?
2. Does the service need to serve requests very frequently?
3. Is the architecture rather monolithic instead of built out of small, loosely-coupled parts?
4. 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?
5. Are processes mostly revolving around synchronous operations?

The more questions answered with **no**, the better. As you probably already know, the use cases are directed at micro-service architectures (especially those built on asynchronous operations). What's also very common and works very well with FaaS are operational tasks that need to be done regularly like renewing certificates via LetsEncrypt.

## Limitations

---

A single Lambda function has limitations you have to keep in mind while designing your target architecture.

- **Maximum execution time** - the longest possible execution duration you can configure is 15 minutes. If this is reached, your function **will be** terminated forcefully. If you're planning for long-running processes, you have to split them up. So running a consistency check on your customers' data which takes quite a while, needs the use of a fan-out pattern to split the running check into multiple instances that are checking only a subset of your data.

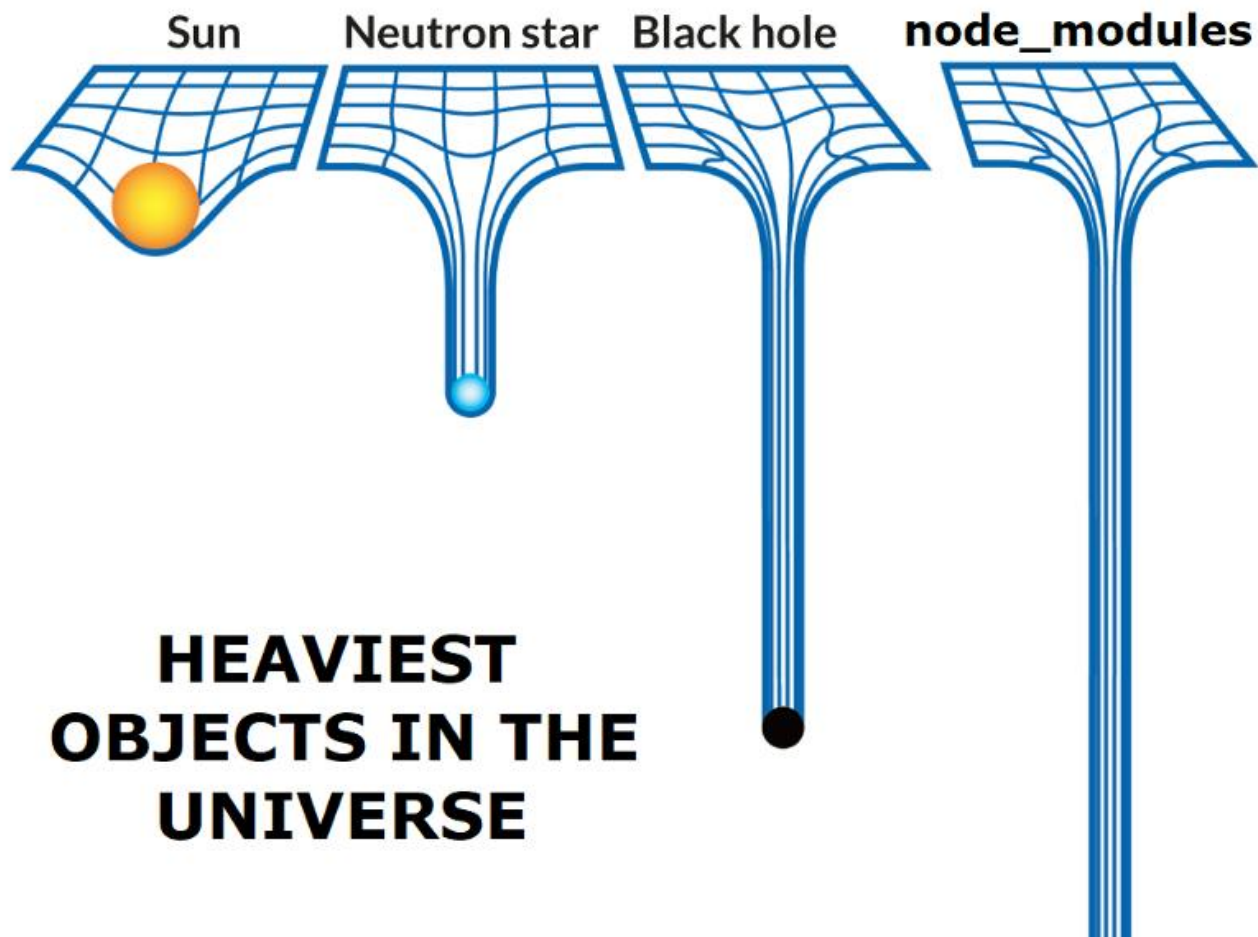
- **Limited built-in storage** - you're only able to use 500MB of temporary storage for your Lambda function. If you need more space, which should probably also be persistent exceeding the lifetime of a provisioned function, you can integrate with AWS' Elastic File System (EFS).
- **Deployment package size** - your unzipped distribution, containing your code and dependencies, can only be up to 250 MB. This also includes the size of all referenced Lambda Layers.

Those are not the only limitations. We've covered this in another article in our blog, where we go into depth about [Lambda's soft and hard limitations](#).

## Lambda Layers and EFS Integration

---

Deploying Lambda functions if you're using Node can be very time-consuming as you need to package your whole distribution containing all dependencies in your `node_modules`.



In general, updates to the dependencies are happening rarely in comparison to updating your business logic or APIs and therefore your code. Also, if you handle multiple lambda functions because you've split your APIs hastily, your functions will require more or less the same dependencies. This brings us to **Lambda Layers**, an extension that helps you manage

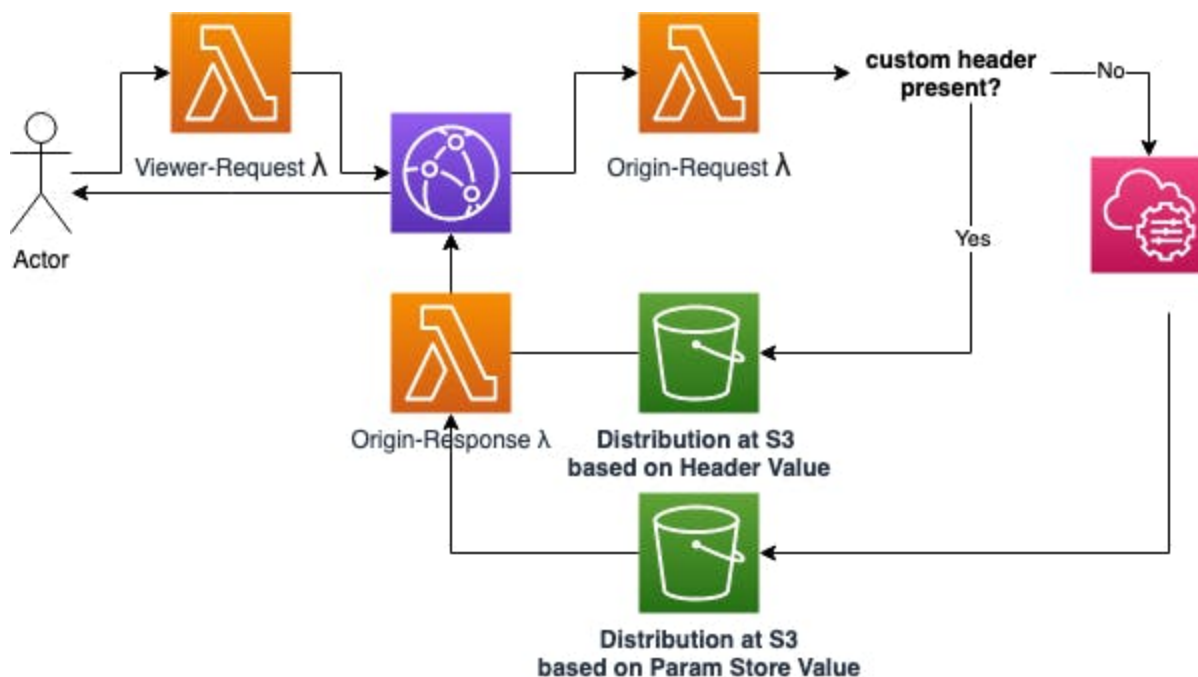
your runtime dependencies and drastically speeds up packaging and deployment processes. A Lambda layer is a packaged folder that can contain all your needed dependencies. You can attach multiple Layers to a single Lambda function and one layer to multiple functions (n:m relationship).



**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`.

## Working with Lambda at the Edge

Lambda is also natively integrated into other services like CloudFront. With Lambda@Edge you can easily achieve otherwise difficult-to-implement functionality. AWS has many Edge Locations all around the world. Dynamic stage routing as an example enables you to route your request to different S3 buckets based on request parameters like headers or query parameters.



The example above shows how you can easily route to different origins with the help of Lambda@Edge.

- **Viewer Request** - We're checking whether we have a current cookie. If the cookie is there and contains our latest version in SSM, we'll forward the request as is. If the value differs, the cookie gets updated and CloudFront won't return any cached value but forward it to the Viewer Origin Lambda.

- **Origin Request** - the function checks for our custom header and assigns the source accordingly. If the custom header was found, we're setting the target bucket as defined by the header at `request.origin`. If not, we're setting the bucket which is saved at the Parameter Store.
- **Origin Response** - will set the Cookie value in the user's browser, so CloudFront will cache correctly for future requests.

## Provisioned and Reserved Concurrency

---

You can take further steps on the configuration of your lambda function by setting up provisioned or reserved concurrency.

- **Reserved Concurrency**—set limits to the maximum number of parallel executions of your lambda function. As your default account limit is 1000 parallel executions, you want to make sure that customer-facing or other business-critical operations always have enough reserved instances.
- **Provisioned Concurrency**—reduce latency fluctuations by having continually provisioned lambda instances, which come with higher pricing.



### Provisioned Concurrency does not completely remove your Cold Starts! AWS'

provisioned concurrency will remove some underlying latency that is caused by provisioning your function, but not the actual bootstrap process for your code. So if you're using some framework like NestJS, spinning up your controllers will still take the needed time when your function gets a cold invocation.

## Security

---

IAM roles and policies protect your Lambda functions. By default, there is no ingress traffic possible from the internet to your function but all egress.

## Attaching your Function to a VPC

---

You can assign your Lambda functions to a private VPC to gain additional security if you need to access resources that are located inside a VPC or if you want to restrict internet access for the function itself. Before AWS introduced Hyperplane support for Lambda, this came with heavy limitations and downsides.

- **Limited ENIs**—your Lambda instance needs an ENI (Elastic Network Interface) at provisioning time if it is assigned to a VPC that has a soft limit per region. Additional notes can be found in the [AWS Documentation on Lambdas in VPCs](#).

- **Limited IPs**—each ENI needs a private IP from the associated subnet. Those are limited by the size of the corresponding CIDR block.
- **Internet access**—if your function needs access to the internet or other AWS services that are not in your private VPC, you need a NAT gateway which introduces additional costs.
- **Slower cold starts**—due to the ENI requirement, cold starts to take significantly more time.

AWS resolved those issues with Hyperplane so that you don't need to worry about ENI bootstrap or exhausting IPs anymore.

## Risks of Lambda's Scalability

---

As Lambda can be scaled out horizontally very quickly, you should be careful about functions that are internet-facing.

- **Usage plans** - enables you to protect your gateway via API keys. Also, you're able to set rate limits for a defined period, the maximum number of requests per second as well as some burst limits for short periods. Usage plans are free of charge.
- **Reserved concurrency** - you can set limits to the maximum number of parallel executions of your lambda function. Keep in mind that if AWS can't take care of a request by assigning a new lambda instance because you're currently already serving the maximum configured parallel executions, the invocation will be rejected.

## Working with Infrastructure-as-Code Tools

---

There's a huge toolset that helps you to provision and manage your infrastructure evolving around Lambda functions with high maintainability and low error-proneness. We also have an introduction to common IaC tools [here](#).

## Serverless Framework

---

The famously known [Serverless Framework](#) adds an abstraction layer on top of CloudFormation. To show the ease of configuration, the following example will provide a Lambda function with an API Gateway with an attached usage plan & an active API key.

```

service: myservice

provider:
  name: aws
  apiKeys:
    - f658e3dc-6449-ebcc-8064-ee1cdfd2ef4f usagePlan:
  quota:
    limit: 1000000
    period: MONTH
  throttle:
    burstLimit: 250
    rateLimit: 10

functions:
  payment-lambda:
    package:
      artifact: myservice.zip
    handler: lambda-proxy.handler
    name: myservice
    runtime: nodejs12.x
    memorySize: 256
    timeout: 10
    events:
      - http:
          path: /myservice/{proxy+}
          method: any
          private: true`

```



**Is your desired resource not yet available in Serverless Framework?** As your stack is managed via CloudFormation, you're also able to extend your YAML files with native CloudFormation code. This helps you to continue if you're stuck on a requirement that is probably not yet implemented by Serverless Framework, but available with CloudFormation.

## Terraform

---

Terraform by HashiCorp is the most famous and widely used tool to manage your cloud infrastructure via code. It's progressing fast and covers nearly everything, also has a large community that is developing different providers for all kinds of services.

We've also written an [extensive introduction guide to Terraform](#).

## CloudFormation

---

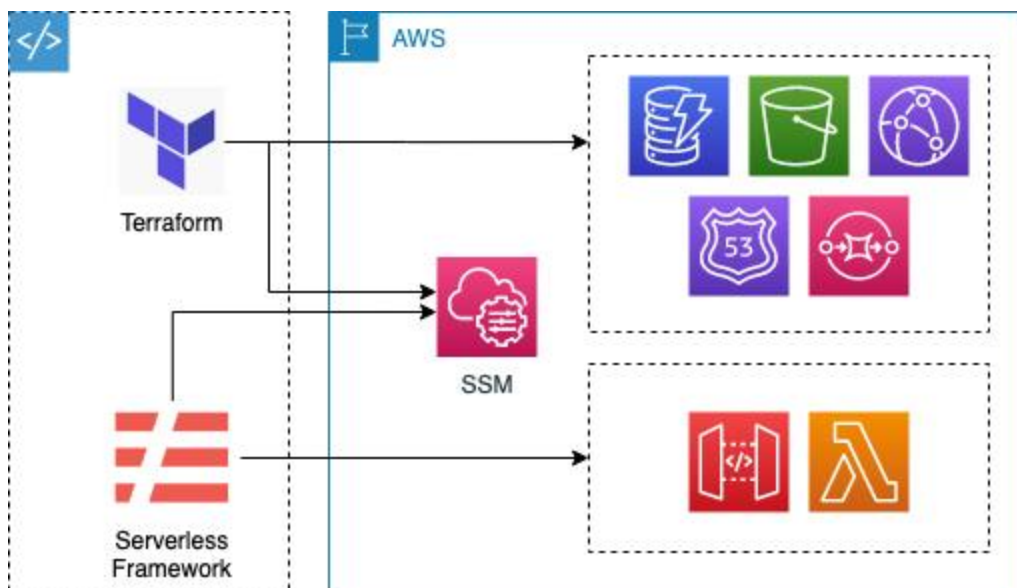
CloudFormation is a native AWS Service designed to manage an application stack. The infrastructure can be defined as YML template files, similar how to it's done with the Serverless Framework. I prefer to go with Terraform over CloudFormation, as in my option



the readability, as well as the possibilities, are much more extensive. But CloudFormation Stacks with Lambda-backed resources are still really nice.

## Working with Multiple Tools

Serverless offers a great toolbox to speed up your development regarding serverless architectures, but in general, you have other infrastructure as well which you probably don't want to manage via custom CloudFormation resources in your Serverless YML file. Serverless Framework is most likely the best choice for Lambda & API Gateway, but Terraform is your all-around tool for everything else, you can combine both via AWS' Parameter Store.



### Combining Terraform & Serverless Framework via the Parameter Store

The steps to take are quite simple:

- **Apply your infrastructure with Terraform**—Terraform will create all your non-Lambda-related infrastructure.
- **Export ARNs of your resources to the Parameter Store**—Terraform can also take care of this step.

```
resource "aws_ssm_parameter" "user_table_arn" {
  type      = "String"
  description = "ARN of the user table"
  name      = "user-table-arn"
  value     = aws_dynamodb_table.user_table.arn
  overwrite = true
}
```

**Use your exported parameters in your serverless YML file** — easily retrieve your exported ARNs so you can use them within your Serverless template file.

```
custom:
  userTableArn: ${ssm:user-table-arn}
```

**Run Serverless deploy** —use Serverless to create & deploy your CloudFormation stack.

## More Learnings and Advanced Tips

---

As I started to design, build, and deploy my first serverless architecture while I knew almost nothing about AWS itself, especially nothing about serverless architectures, I encountered a lot of obstacles on the way but also gathered a lot of learning.

### Asynchronous Executions within Your Function

---

I'm heavily using Node for my Lambda functions and I stumbled on weird behaviors I could not explain in the beginning. One of them was entity modifications in DynamoDB which were executed after the response had been sent to the client. The reason for that was that I was not using `async` and `await` consistently. Sometimes, the last modification at the end of the request was done without `await`, so the modification was sent to the event loop, but the Lambda function itself froze immediately afterward because the response was sent to the client and the end of the execution stack was reached.

If the same function instance was executed at the next incoming request, the modification in the event loop finished. This led to infrequently occurring data inconsistencies. The issue's not really about Lambda, but more about not understanding how Node is working internally.

### Minimizing cold starts for client requests

---

Provisioning your lambda function can take a lot of time, based on your choice of frameworks. I was experimenting with Kotlin and Spring Boot in the beginning, which did not result in any good results for latency requirements. Even if you create a CloudWatch rule to regularly invoke your Lambda function to keep it warm, there is no guarantee that the requests from your customers will hit those instances. Also, multiple incoming requests in parallel can trigger multiple Lambda instances, so keeping one instance "hot" won't help in this case.

Creating a dedicated "warm-up" function that will regularly trigger your Lambdas helps to decrease the number of encountered cold starts, but it won't help to get rid of them completely. That's why the most important thing is to keep the time, which is needed by your application to start, as low as possible.

Finally, I switched from Java & Spring to Node & NestJS (build on top of Express), which offers really fast start-up times.

## Application Provisioning Outside of Handler Function

---

The referenced entry point for your Lambda function is always your declared handler method. But what's happening with variables and processes that are declared outside of this method? The details on this are quite nice to know:

- **will be executed at first**—all of your globally defined lines are executed before the handler gets invoked, but only when your function is provisioned.
- **not destroyed after execution**—as long as your function stays provisioned, all objects will be retained.



**Function errors:** If your function exits prematurely with a non-zero exit code, the next execution will always be a cold start.

- **excluded from payments**—as long as it does not take more than 10s, you **won't** be billed by AWS for the execution time. Yeah, I'm not joking on that one.
- **running at max capacity**—your global code will run at the **highest** Lambda specs, even if you configured your Lambda function with the lowest vCPUs and memory (= 128MB).

Make use of this.

```

import { Server } from 'http'
import { Context } from 'aws-lambda'
import * as serverlessExpress from 'aws-serverless-express'
import * as express from 'express'
import { Express } from 'express'
import { createAndConfigureApp } from './app-config'

let lambdaProxy: Server

async function bootstrap() {
  const expressServer: Express = express()
  const nestApp = await createAndConfigureApp(expressServer)
  await nestApp.init()
  return serverlessExpress.createServer(expressServer, null, null)
}

bootstrap().then(server => lambdaProxy = server)

function waitForServer(event: any, context: any) {
  setImmediate(() => {
    if (!lambdaProxy) {
      waitForServer(event, context)
    } else {
      serverlessExpress.proxy(lambdaProxy, event, context)
    }
  })
}

export const handler = (event: any, context: Context) => {
  if (lambdaProxy) {
    serverlessExpress.proxy(lambdaProxy, event, context)
  } else {
    waitForServer(event, context)
  }
}

```

## Single Function Deployments

---

What also helped me to gain a lot of development speed is a small bash script to package a specific function and update it immediately via the AWS CLI. With this, you don't have to run Terraform or Serverless Deploy, which takes a significant amount of time.

This is also possible with Serverless via `serverless deploy function --function <function-name>`.

## Final Thoughts

---

Lambda offers a great way to spin up a new service fast and helps to keep operational burdens low. Also, you can automate a lot of reoccurring tasks hastily without having to think about patching the OS or worrying about failing containers.

I'm using Lambda in all kinds of ways for a lot of different production services and I will continue with this in the future. In addition, AWS is improving and extending Lambda regularly, making it even better.

**Published on**

---