AWS LAMBDA

RUN CODE WITHOUT THINKING ABOUT SERVERS OR CLUSTERS

INTRODUCTION 👋

Virtual machines and containers made infrastructure management a lot easier. Lambda takes this another step further by virtualizing the runtime and only offering functions as a unit of scaling. As you're not responsible for any infrastructure, this is called **serverless**.

You'll only bring the code and the Lambda environment takes care of provisioning the underlying infrastructure and micro-containers to execute it.

The Evolution of Serverless

Virtual Machines → **Containers** → **Functions**

LAYERS 🔶

Necessary dependencies need to be included in your deployment package. As dependencies can quickly reach multiple megabytes, packaging and code deployment can take up more time, even if you just want to update your own code.

With Lambda Layers, you can can bundle your dependencies separately and then attach them to **one** or **several functions**. You can also use multiple layers in the same function.

Be aware, that your Lambda deployment unit and all of the attached layers can't exceed 250 MB in an unzipped state.

SECURITY 🔑

Your function is protected via IAM. By default, there's no ingress traffic possible to your function, but all egress to the internet.
You can attach your function to a VPC to access other private resources or restrict egress traffic to the internet via security groups and network access control lists.

MICRO-CONTAINERS 🤖

Don't get this wrong: serverless doesn't mean that there are no servers at all. It's just abstracted away and managed by AWS.

For a function invocation, AWS will spin-up a micro-container if there's none that is already available for your function. The container is kept for a small period of time until the resources are freed so they can be assigned for other tasks.

RESERVED CONCURRENCY 🌟

Reserved concurrency ensures that that is concurrency level is always available to your function.

This means:

1. The reserved concurrency will be subtracted from your accounts regional soft limit. This limit is shared by functions that do not have a reserved concurrency configured.
2. Your function can't exceed the concurrency. If you've configured a reserved concurrency of five, the **sixth parallel request** will result in an invocation error.

LAMBDA@EDGE ⚡

Lambda's not only good for computation workloads or REST backends. You can also use them with CloudFront.
It enables you to execute code at different stages when your CloudFront distribution is called.

By that you can for example easily implement authorization rules or route calls to different origin servers.

Generally, you can do a lot as you're also able to **use the AWS-SDK** and **invoke other services**.
💡 Another tip: **CloudFront functions** - the lightweight, cheaper alternative with a reduced feature set.

COLD STARTS 🥶

Starting a micro-container on-demand takes time. This is called the **cold-start** period. Lambda needs to download your code and start such a small container to run it. Additional time can be taken to bootstrap global logic of your function, e.g. loading your main framework.

There are a lot of best practices and tricks to reduce the time until your actual function logic can execute:

- Keeping the packaged size of your functions small.
- Regularly health checks to invoke your functions.
- Bootstrapping general code outside the handler function.

PROVISIONED CONCURRENCY 💨

Reduce your latency fluctuations by keeping the underlying infrastructure provisioned.

Keep in mind: this introduces **additional** costs.
Also, deployments of your function will take more time as your updated code needs to be published as a fixed version.

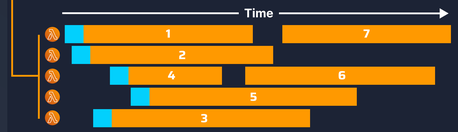
SERVERLESS BENEFITS 🧡

We believe that cloud-native development like serverless computing are the future as they come with great benefits:

- **reduced operations** as you don't need to worry about infrastructure.
- **out-of-the-box scalability** for your functions.
- a **pay-as-you-go** pricing model - you won't pay for unused compute resources.
- **higher agility & development speed** compared to traditional approaches.
- a lower entry barrier to **event-driven architectures** due to native integrations with messaging services like SQS and SNS that enable you to build very resilient ecosystems.

THE INTERNALS ⚙️

Each provisioned Lambda micro-container is only able to process **one request at a time**. Even if there are **multiple containers already provisioned**, 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 started due to the request timings. The **first re-use did only happen at request number six**.

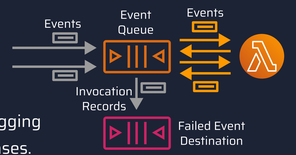
VERSIONING & ALIASES 📸

You can publish immutable versions for your Lambda functions and then use a reference at all places that need to invoke this functions.

This allows you to easily switch between different function versions. If your function is exposed to the internet via AWS API Gateway, you can also integrate canary deployments by using weighted aliases. This enables you to send only a fraction of traffic to your new deployment to check for error spikes while the major load resides on your stable version.

DESTINATIONS 🔀

If your Lambda function is invoked but fails to complete successfully, you can use the Lambda destination feature to send the results of the failed invocation to a different destination, like a SQS queue. This can be useful for debugging or for handling error cases.



THE HANDLER METHOD 🏁

What's important here: everything **outside** this entrance method will be **executed first** when your function receives a cold start and **won't disappear** from memory until it's de-provisioned.

The execution of the global code outside of your handler method will be executed with **high memory & CPU settings** and **isn't billed** for the first 10s.
Make use of this by always bootstrapping your core framework outside the handler method.

OBSERVABILITY 🔎

Having in-depth observability for your lambda-powered, event-driven architecture is still a goliath task.
Mostly, you'll design event-driven, async architectures that involve a lot of **other services** like for example SQS, so there's just not a single log group to monitor.

CloudWatch helps a lot in the first place via Metrics & Alerts. Many of them are **predefined**, like:

- **Errors**: your function did not finish with exit code zero
- **Throttles**: concurrency limit was exceeded

Familiarize yourself with CloudWatch's possibilities.

For enterprise-scale applications: rely on third-party tools like **Dashbird.io** to get all activities of your Lambdas and other related, common services like API Gateway, SQS, SNS, DynamoDB & much more in a **single place**.

SERVICE EVOLUTION 🌔

Lambda is continuously improved. Since Lambda's release date back in 2014, AWS introduced among other things:

- AWS Hyperplane support for Lambda to **significantly reduce cold start times** when attached to a VPC.
- fine-grained billing in **1ms periods** instead of 100ms.
- running Lambda on **Graviton2 / ARM** processors, which offer a better cost-performance efficiency.
- much **higher memory & vCPU configurations**.
- up to **10GB of temporary local storage** at /tmp.

aws