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.

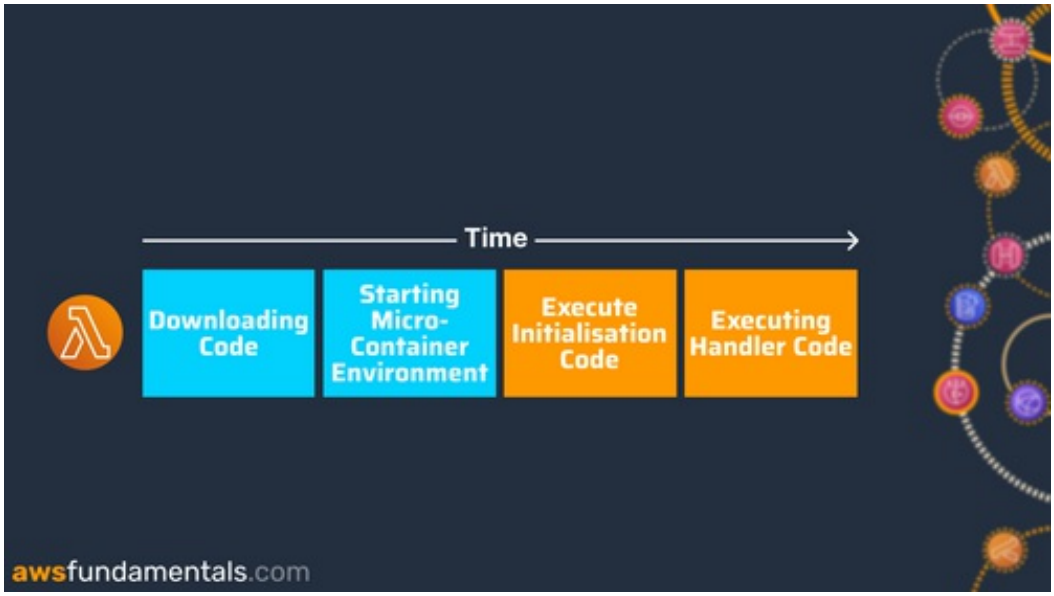
For an incoming request to your Lambda function, AWS will either

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.

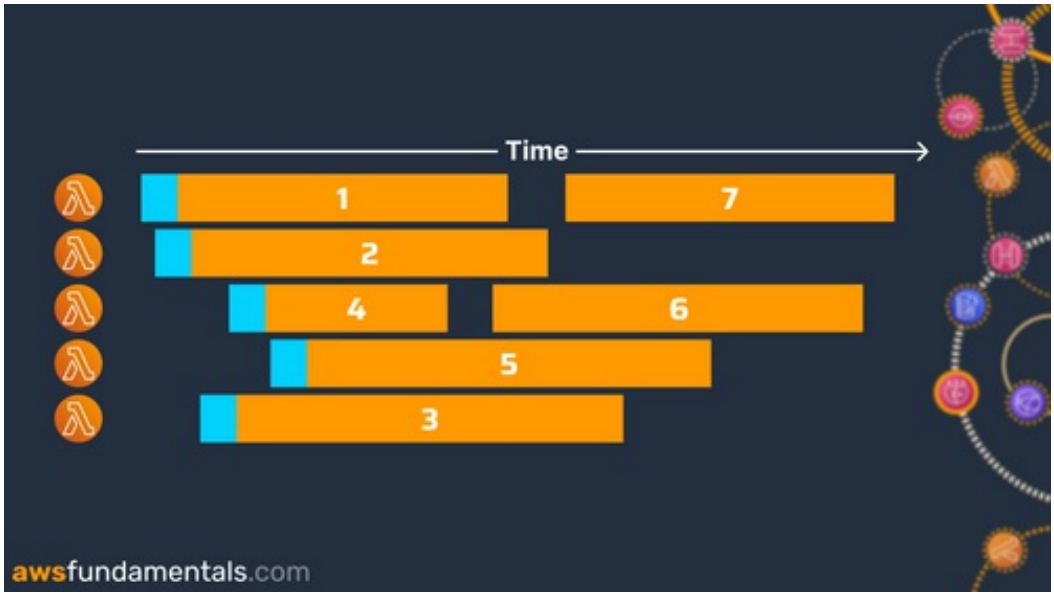
Lastly, your main code is run that's inside your handler function.

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.

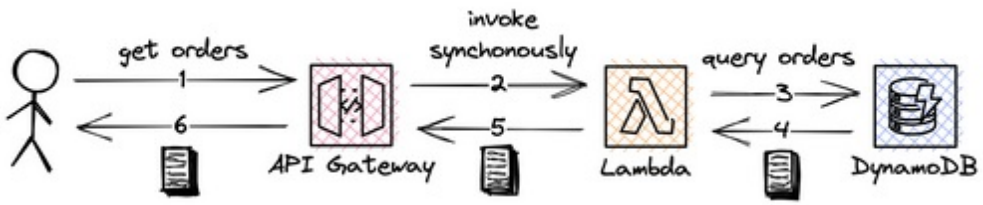
Make use of this and bootstrap as much as possible outside the handler function and keep a global context while your function is running.

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

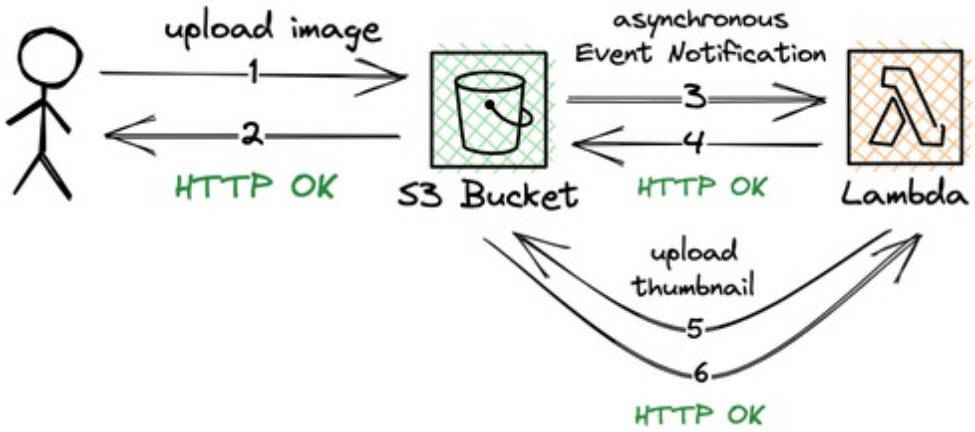
There are two methods to invoke your functions code:

- **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.



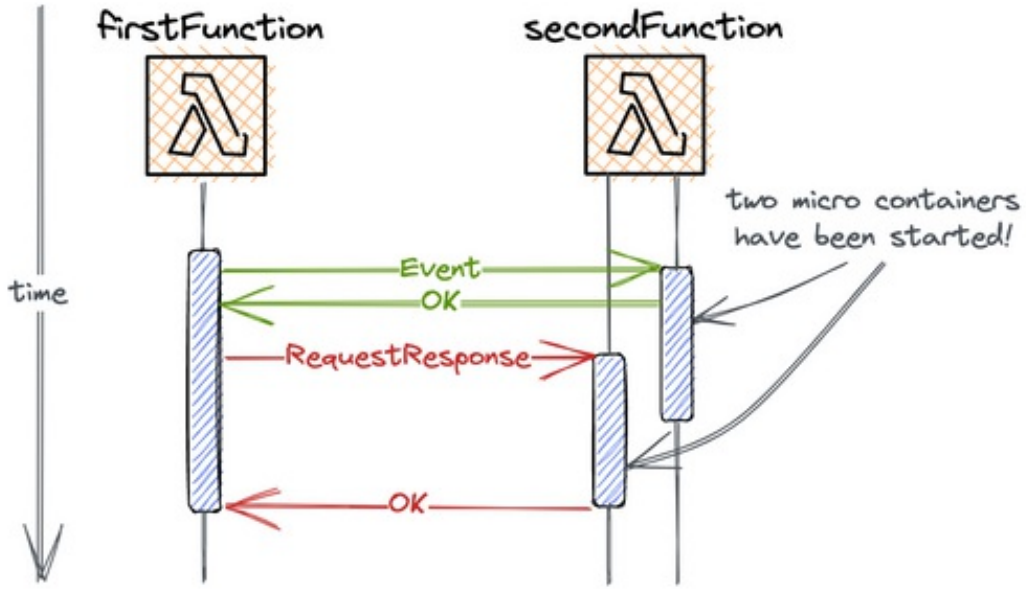
If you're invoking functions from another place, e.g. another Lambda function, the invocation type depends on how you want to handle results. Synchronous invocation is useful when you need to retrieve the result of the function execution immediately and use it in your application.

```
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.

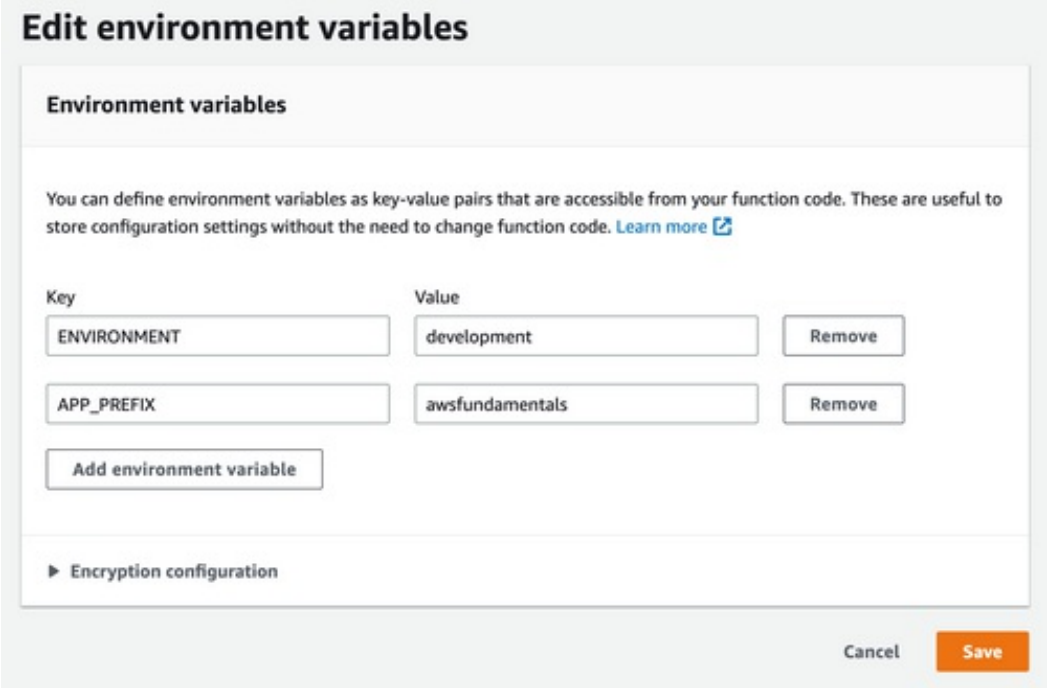
As with other services, it is a good security practice to create an execution role with the **least privilege.** This means it should only have the permissions that are required for the function to perform its intended tasks. This helps to reduce the risk of unintended access to resources and data.

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.

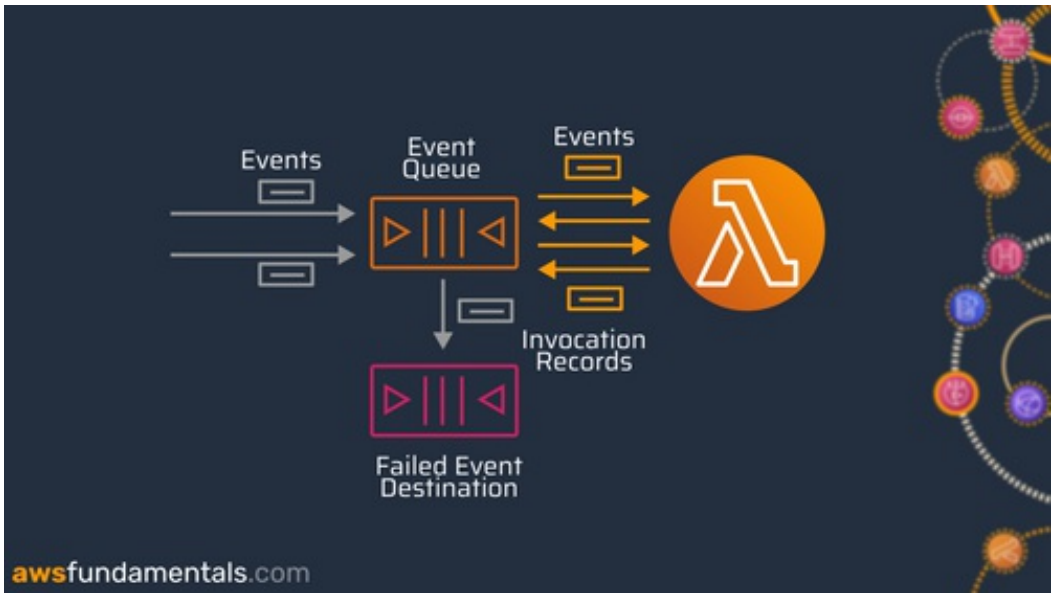
Prominent examples are:

- 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

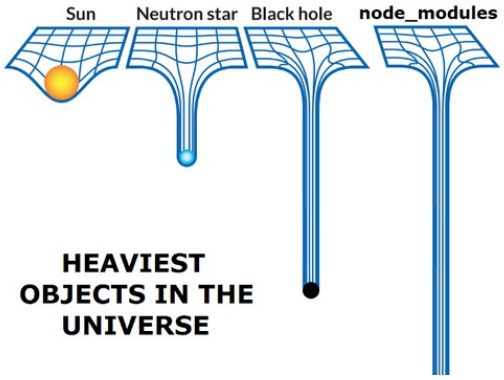
Regardless of the strategies you're using of keeping Lambda functions warm via strategic health checks or your level of permanent requests per second, your micro-containers **will be de-provisioned** at some time.

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

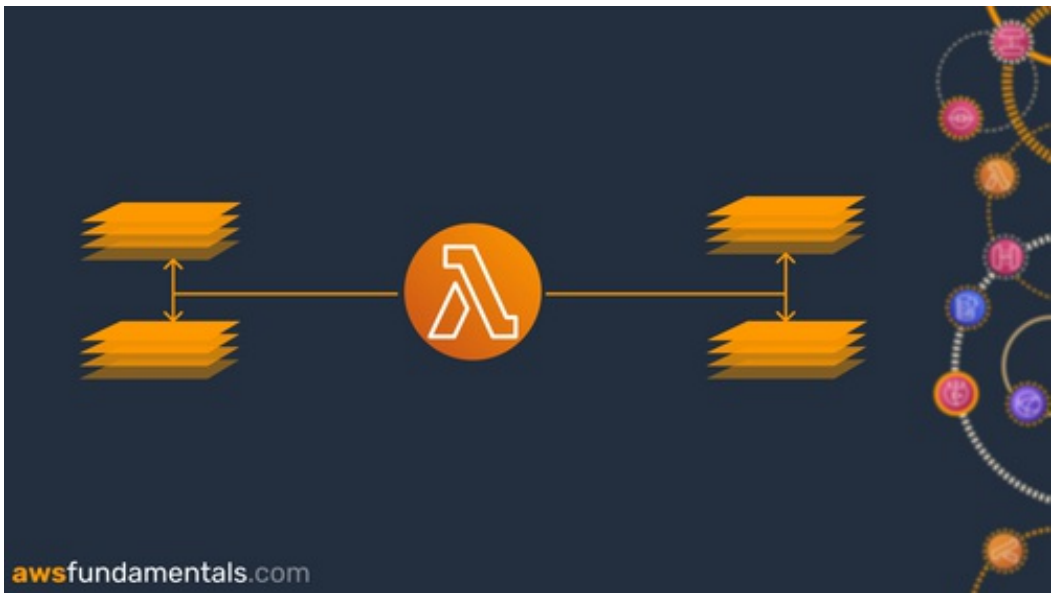
Building extensive business logic often doesn't require you to reinvent the wheel, but to make use of existing libraries. This mostly results in having more code in dependencies than in actual self-implemented business logic which will slow down packaging and deployments.



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

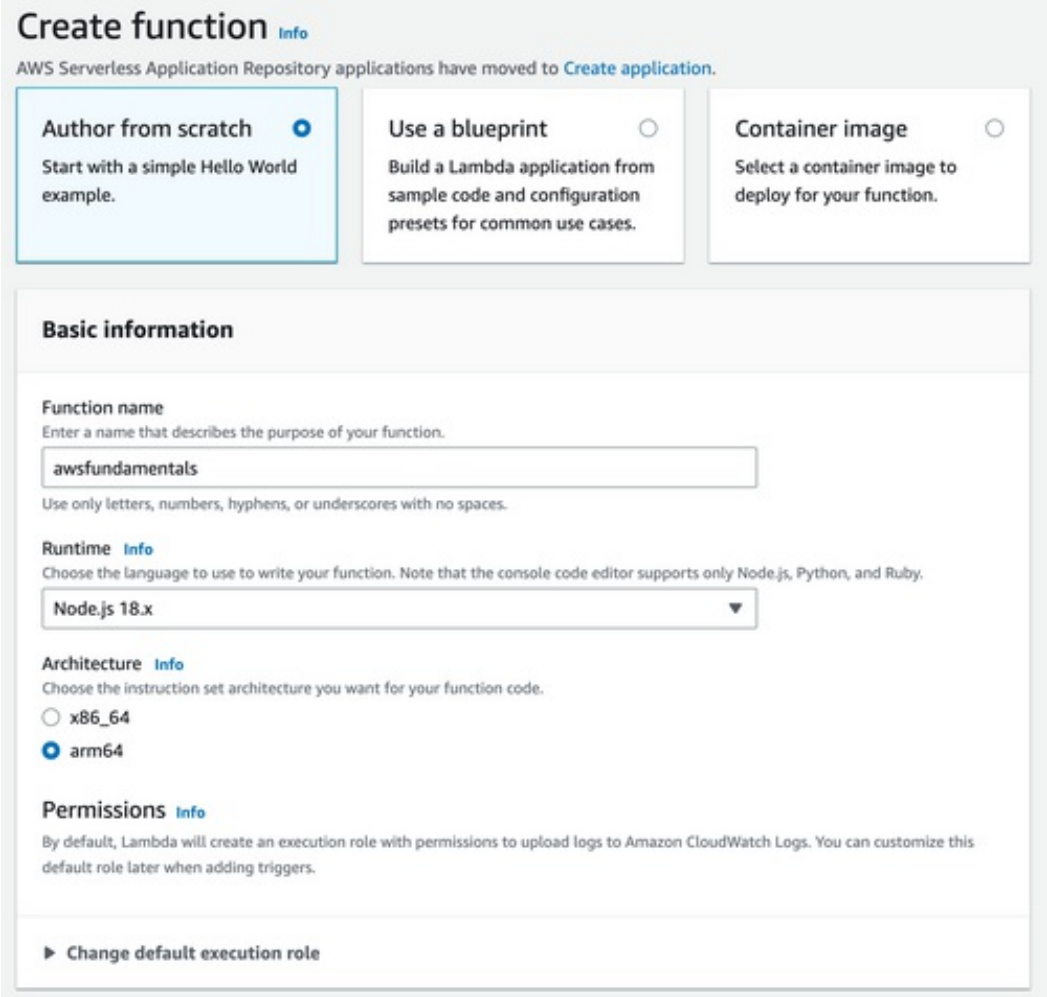
We've gone through the most important fundamentals of Lambda. Let's jump into the doing and create our first, own small Lambda project.

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

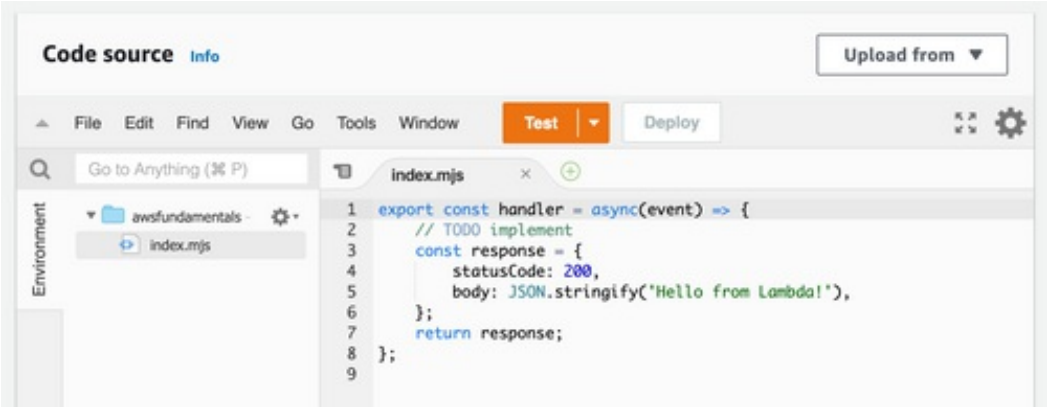
1. **Creating a simple Node.js function**. We'll create a small function, and adapt and deploy code changes within the AWS management console. We'll also test our function here via our own test events.
2. **Adding external dependencies**. We'll add Axios as a dependency so we can execute HTTP calls in 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

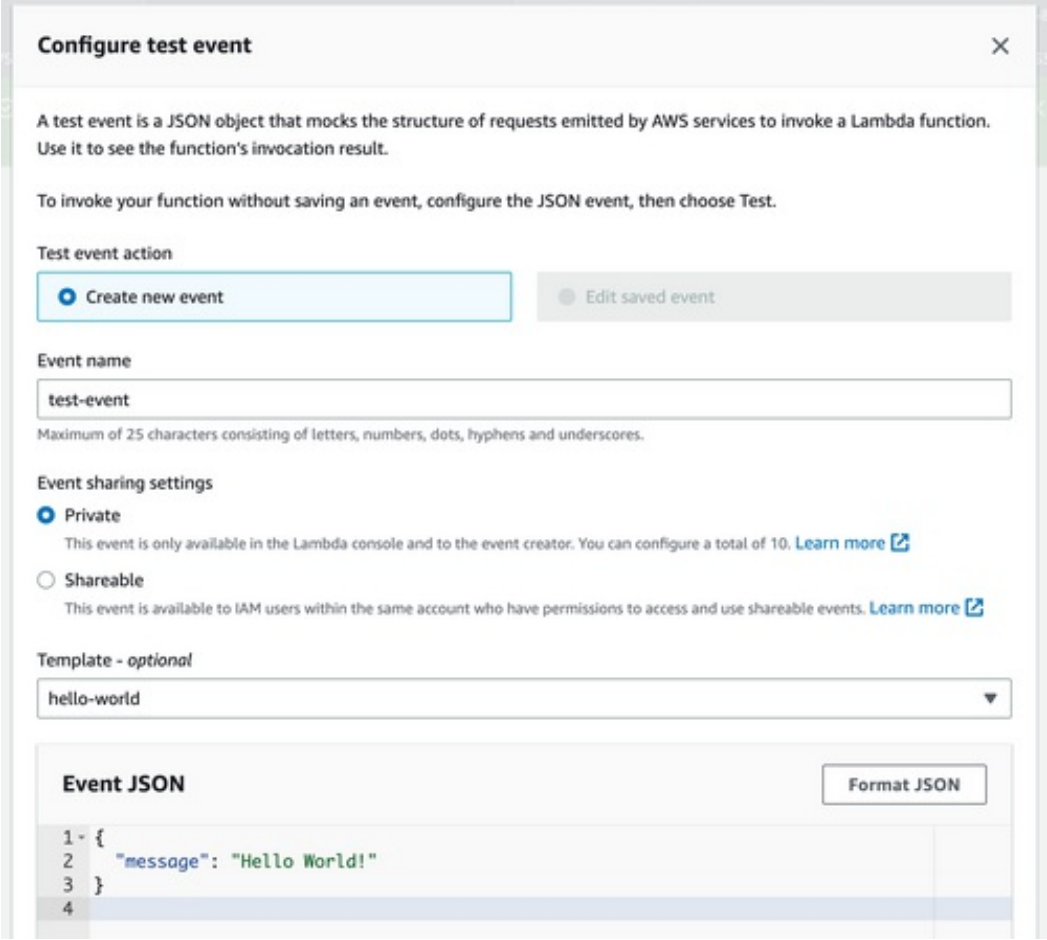
Jump into the AWS Lambda console and click on `Create function`. We only need to define a name, select our target architecture, and chose which runtime we want to use. For our example, we'll go with Node.js.



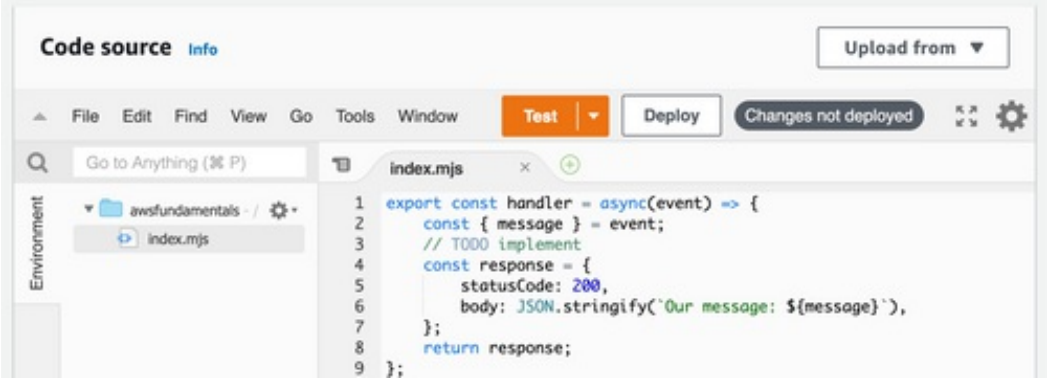
After clicking create, you'll be taken to your functions overview. You'll immediately notice the live editor for our function's code. This editor can also be used to test our function and to save and deploy updates to our function.



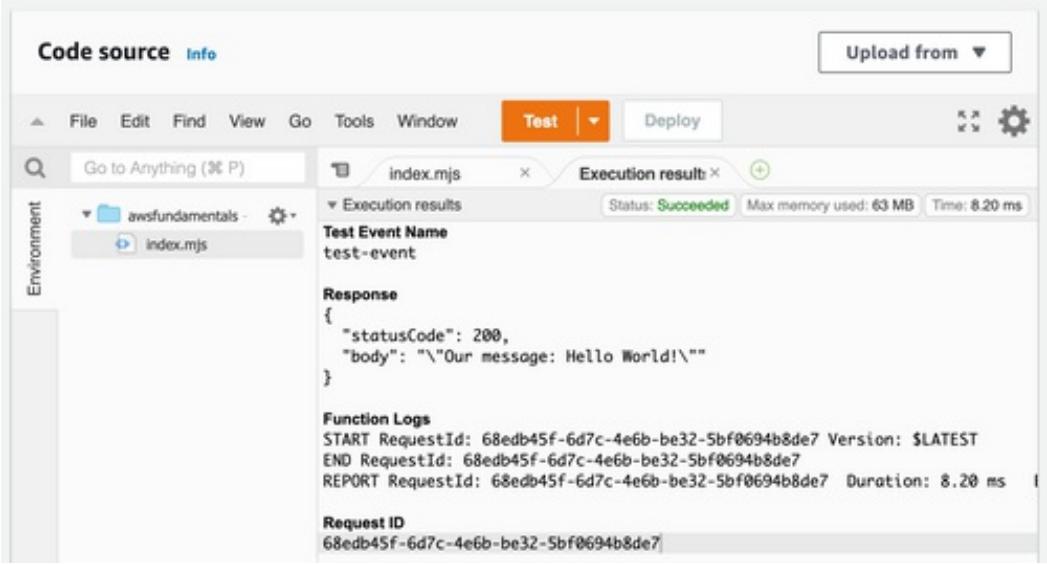
Let's do exactly that by clicking on `Test`. This will open a modal where we can create our first test event. Let's pass a JSON object with a field `message` to our function.



Now let's adapt our function and return the message we pass in the function's response. After clicking `Deploy` the changes will be deployed to our function. Now we can invoke our function with the test event.



We'll get what we expect: our message!



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

Adding External Dependencies

The first challenge was solved. Let's continue and add some external dependencies. We'll initialize a new project via `npm` and install Axios. We'll also add 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 a little bit by adding a new HTTP call via Axios to

`https://ipinfo.io/ip` to get 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 our `node_modules` folder into a ZIP archive which we can then upload to our Lambda function via `Upload from > .zip file`.

```
zip dist.zip .
```

After uploading it we can execute it again via the test button. The response should look something like this:

```
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
```

While our function is still very small because we're only including Axios, its size can increase quickly. After a certain threshold, it's not possible to view or edit the code in the Lambda console anymore. We'll also upload a lot of kilobytes for every function upload, even though there are often only changes to our code.

Let's fix that in our next part by extracting our dependencies into a Lambda Layer.

Externalizing Dependencies into a Layer

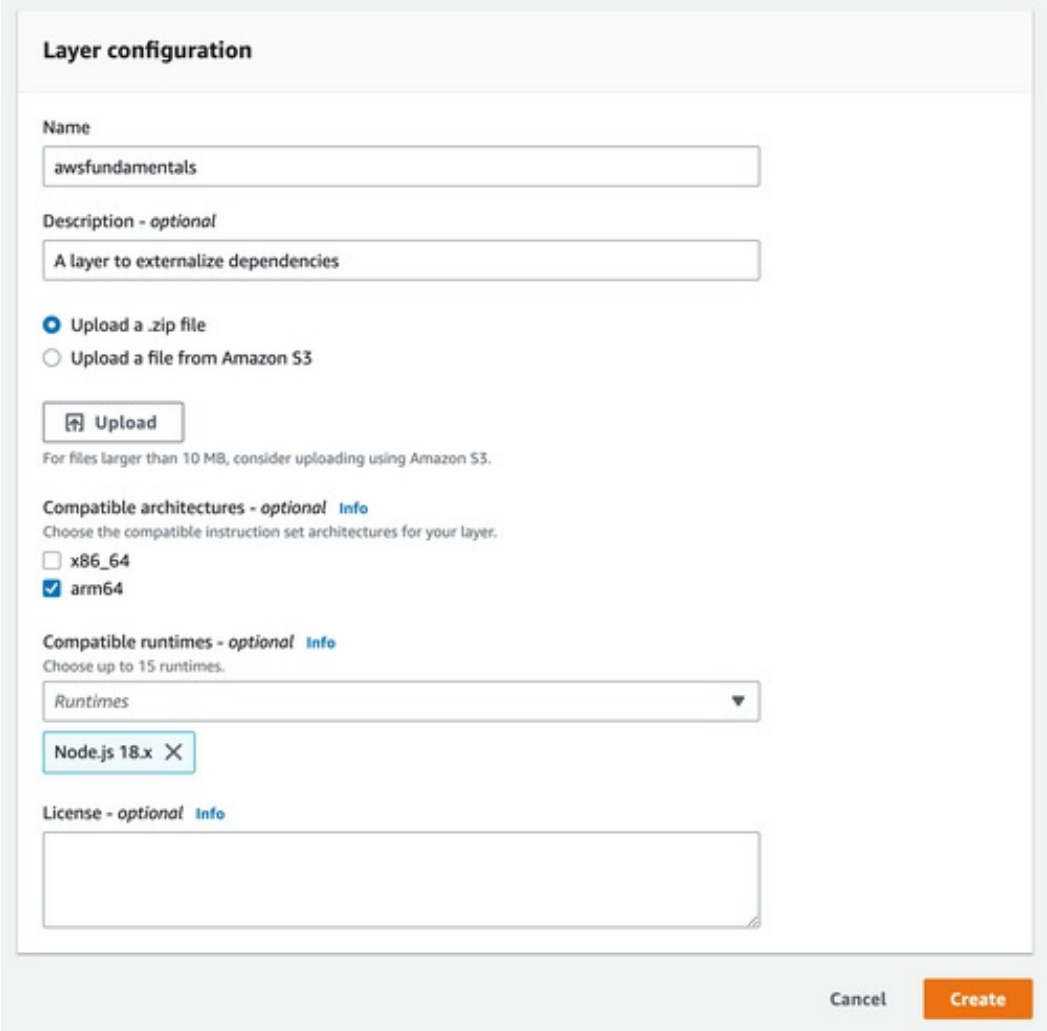
Creating a Lambda Layer does improve two major things:

- we'll reduce the size of the deployment unit we need to upload on function updates
- we can share a layer with several Lambda functions which may all have the same dependencies

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

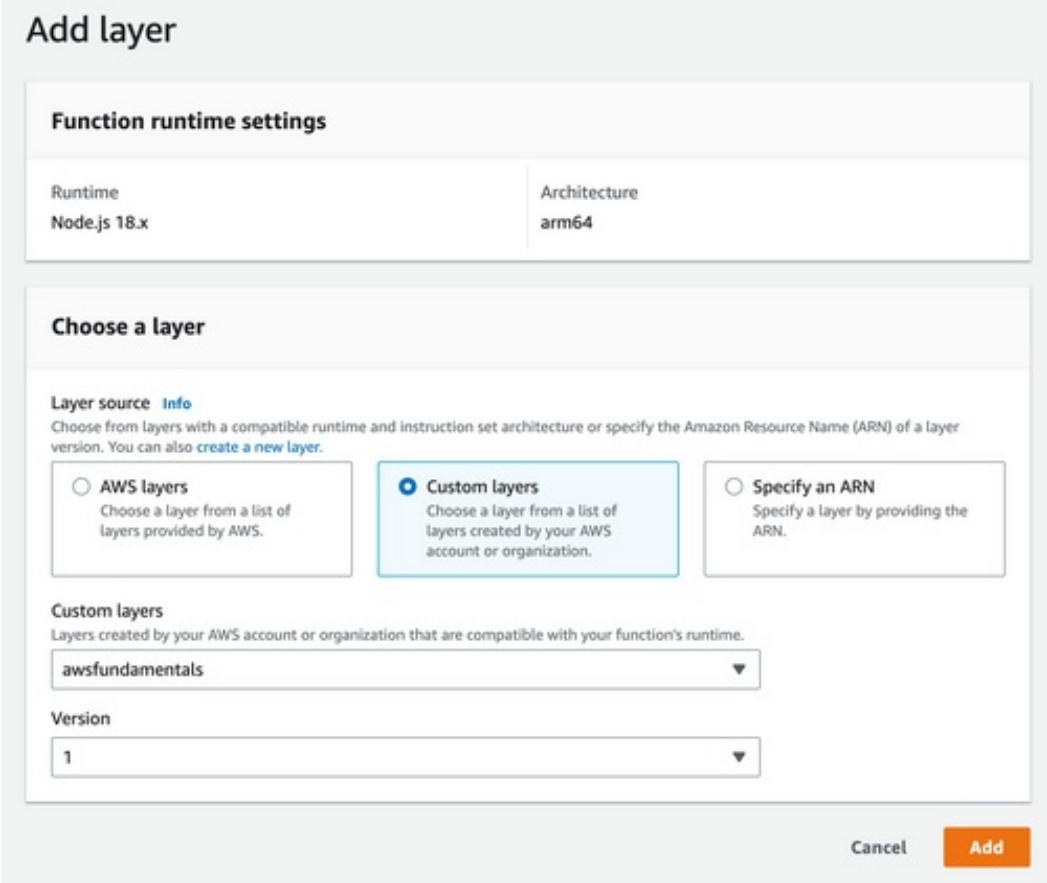
```
mkdir -p nodejs
cp -r node_modules nodejs
zip layer.zip nodejs
```

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



After clicking the `Upload` button, selecting our created `layers.zip`, and finally creating the layer via `Create`, our Lambda Layer is ready to use!

Let's head back to our function and scroll down to the layers overview to connect our new layer view `Add a layer`.



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

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

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

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

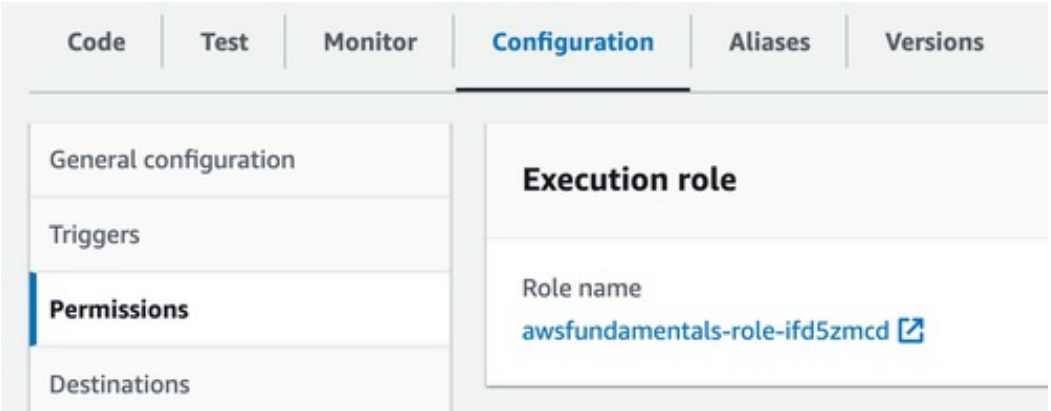
Invoking Another Lambda Function

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

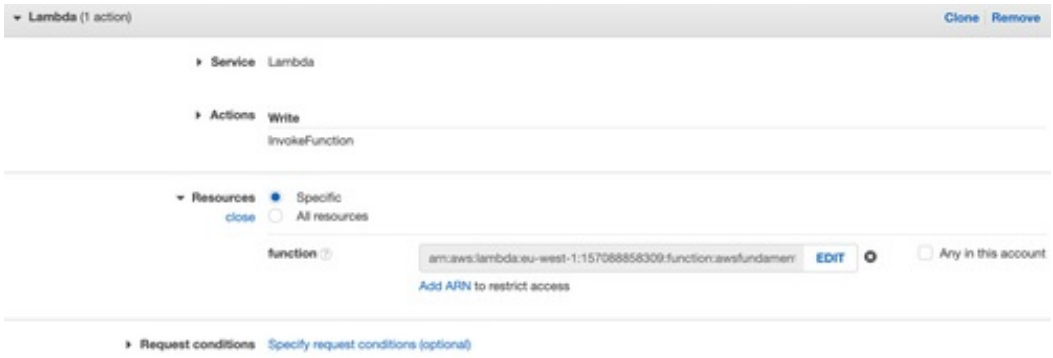
For invoking our function we need two things:

1. our first function has to have the `lambda:InvokeFunction` permission and
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. With a click on the link, you'll be taken to IAM where we can edit the policy.



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



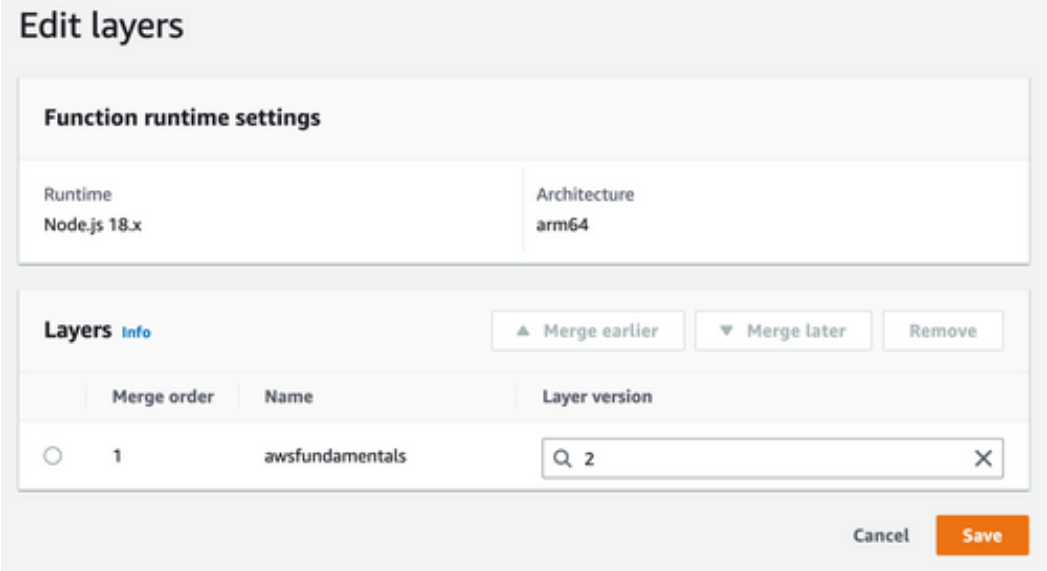
The final JSON policy should now include our new action.

```
{
  "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
```

If you go to your layer, there's a button `Create version` to upload a new version of the layer. Afterward, we can go back to our initial function and switch to our new version `2`.



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've added some waiting periods until the function returns so we can later clearly see the impact of the different invocation types. By default, we'll wait 10 seconds, but we allow passing the value via the incoming event.

Let's head back to our initial function and add the invocation part. For now, we'll invoke the function synchronously.

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

exports.handler = async (event) => {
```

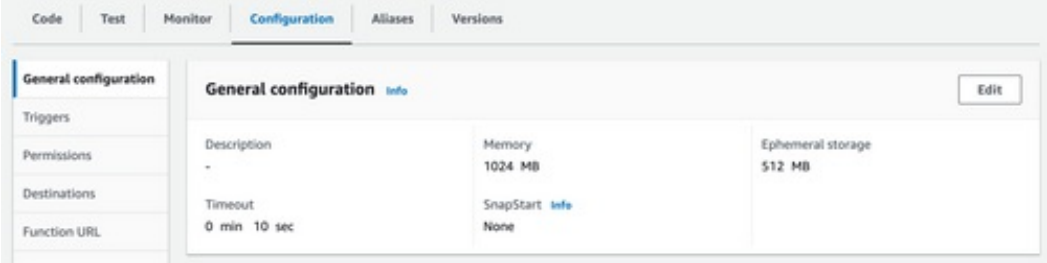
```
  const startTime = Date.now();
  // Invoke the target function synchronously
  await lambda
    .invoke({
      FunctionName: "awsfundamentals-invoke",
      InvocationType: "RequestResponse",
      Payload: JSON.stringify({
        waitingPeriodSeconds: 5,
      }),
    })
    .promise();
  const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

Let's deploy this 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. But looking at our function's configuration, it does. The default timeout for Lambda is 3 seconds. As we wait for 5 seconds until our second function finishes its execution, both functions will time out. Let's adapt the timeout at `Configuration > General Configuration` and set it to 10 seconds.



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 deploying the function again, we'll see in the next invocation that the execution time significantly dropped.

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

Now our initial function doesn't wait until the execution of the second one has finished. The execution is faster, but we don't know if the second function finished its execution without errors.

We've covered a lot in this small project, which is a great starting point to continue playing around with Lambda.

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're able to configure the authentication type to `NONE` to allow public, unauthenticated invocations.

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

Attaching a Shared Network Storage with EFS

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

For a durable storage solution, you can either go for 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 & 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.

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

With CloudFront, you're 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'll get into detail about this in the upcoming chapter about CloudFront - we just wanted to include this here for the reason of completeness.

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're only paying for the actual time at which your code is executed.

What you'll 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's 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.

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 can't be changed unless AWS updates its policies. Let's have a look at the most important ones as it's likely that you'll 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's worth checking back with the current state at AWS Quotas.

A Deep Dive into Great Lambda Use Cases

Due to its on-demand pricing, low entry barrier, and ease of use, Lambda is the most flexible service of all AWS offerings. There's almost nothing you can't use Lambda for. The fact that Lambda natively integrates with a lot of other services, often without writing much or any glue code, also majorly contributes to this.

Consider this list as just very few examples as it is possible to go on for weeks or months just writing about the simple or extraordinary use cases for which Lambda is known already.

Creating Thumbnails for Images or Videos

Classic approaches to video or image processing involve uploading files to storage and having a server regularly pulling for newly created files.

This will introduce costs as your servers also have idle times if traffic is not constant but with periods of high or low traffic.

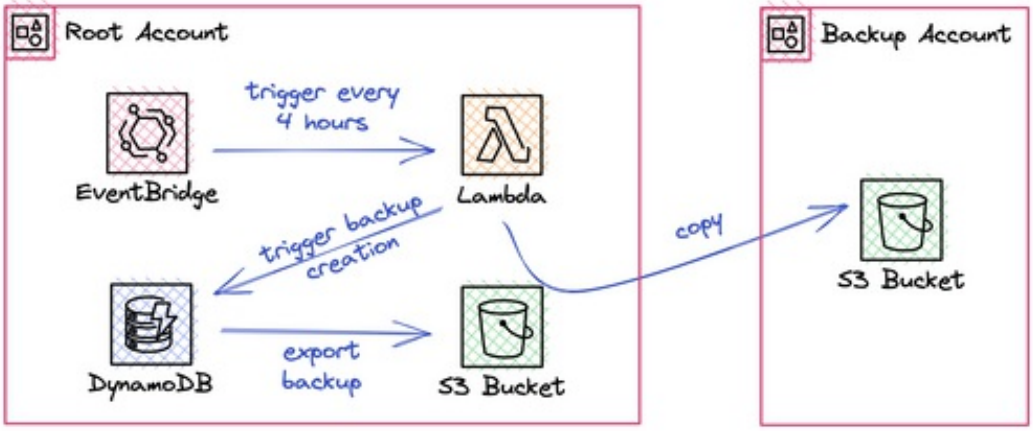
Putting video conversion or thumbnail generation into the hands of Lambda, while uploading

files to S3 gives you a completely different approach. You're able to attach Lambda invocations to S3 object lifecycle events like *ObjectCreated*. With this, new files will automatically trigger your function with an event that contains all necessary information about the lifecycle event and file.

You won't pay for any idle times, as only computations will be billed with Lambda.

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 have redundancy and with that high security.

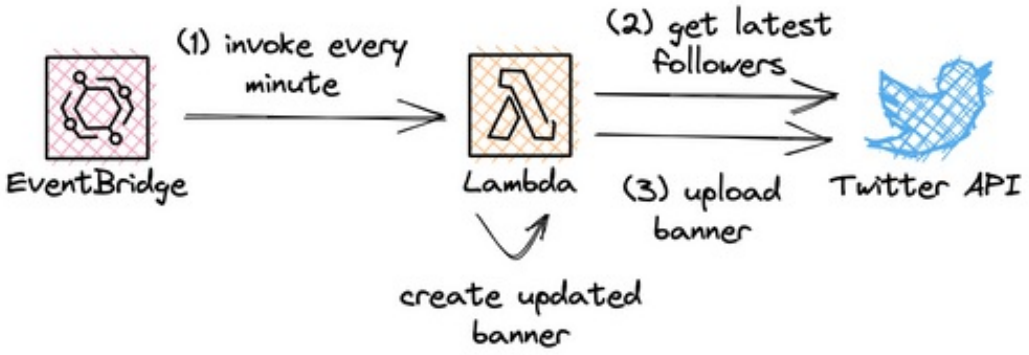


Looking at the previous image and video processing use case: you can also directly synchronize files based on lifecycle events and event notifications.

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 for which you can find a lot of blog posts is to automatically update your Twitter banner based on recent followers, the blog post you've published recently, or any other information you want to show in near real-time.



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.

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 are relying 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 does 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 do 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.

The Best Practices to Reduce Cold Start Times

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

There are a lot of tricks and best practices to reduce cold start times and also 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 does require 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 to 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.

How to Determine If Lambda and the Serverless Approach Is the Right Fit

As mentioned in the starting chapters, we believe cloud-native is the future. This future heavily evolves 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).

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.