Important AWS Lambda Technical Constraints

In this lesson, you will learn about the important technical constraints of AWS Lambda that you should know and assess before deploying your application.

WE'LL COVER THE FOLLOWING

Lambda is still a relatively new service, and it is evolving quickly. I often talk about serverless development at conferences. It's been quite fun to review my slides about problems and constraints and have to remove things every few months. Things like start-up times and Payment Card Industry (PCI) data compliance were a serious limitation in 2017, but then got fixed. A common complaint against Lambda was that there was no service-level agreement guarantee for it, but in October 2018, AWS published an SLA for Lambda as well. (It's currently 99.95%.)

Technical limitations

By the time you read this, the constraints might have changed slightly, but at the time this was written, there were four important technical limitations that you needed to consider when evaluating whether something should run in Lambda:

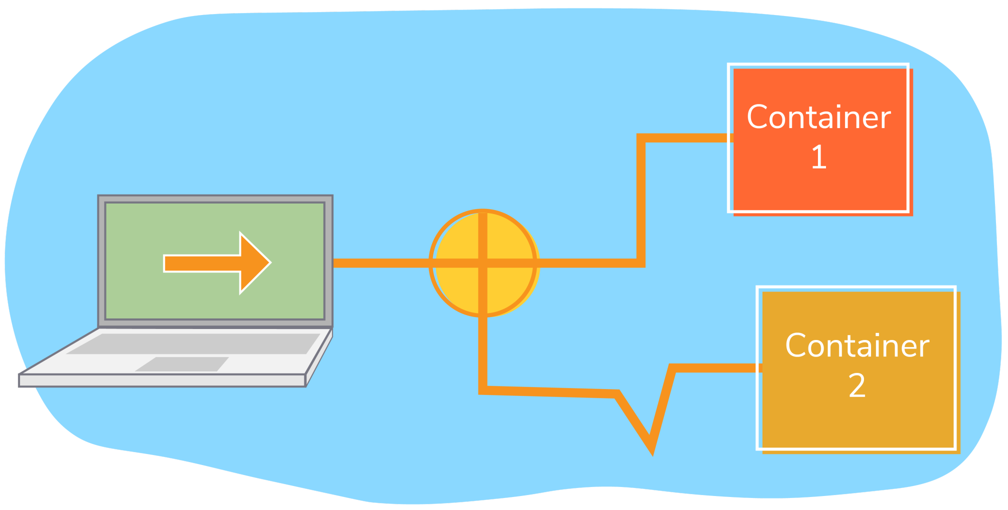
- No session affinity

- Non-deterministic latency

- Execution time-limited to 15 minutes
- No direct control over processing power

No session affinity

Because the hosting provider controls scaling, you generally don't get to decide about starting up or shutting down instances. Lambda will decide whether it needs to create new virtual machines to handle requests, as well as when to reuse an old machine and when to drop it. It might send two subsequent requests coming from the same user to the same container or to two different containers.

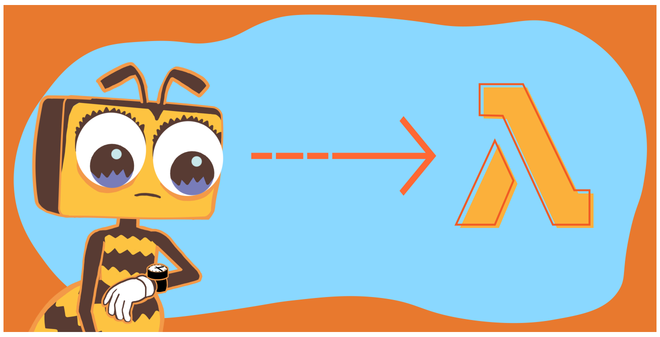


Purely on Lambda, there is no way to control request routing or somehow ensure that requests from the same source arrive in sequence to the same destination. You can achieve sequences and routing control with some other services such as **AWS Kinesis**, but Lambda can still decide at any point to throw away a virtual machine and start a new one.

This limitation, coupled with how functions are named, sometimes causes misunderstanding about Lambda's execution model. Lambda functions are not stateless, at least not in the pure functional programming sense where nothing is kept between two executions. Each instance still has a container with its local memory space and if Lambda decides to reuse a virtual machine, two requests might still be able to share state in memory. Virtual machines also have access to a temporary local disk space, so it's possible to store information between requests on disk. But there are no guarantees about preserving state across requests, and application developers have no control preserving state across requests, and application developers have no control over the routing.

It’s best not to count on any in-memory state between two different requests in Lambda. When designing for Lambda functions, don’t design for stateless execution, design for a share-nothing architecture. You can still cache or pre-calculate things that do not depend on a particular user, but user sessions and state have to be somewhere else. There are several alternatives for session data. They will be explained in Chapter 7 and Chapter 8. In the final chapter, you will also see some typical ways of sharing state or configuration between Lambda functions.

Non-deterministic latency



Lambda is optimized for maximizing throughput, not for minimizing latency. It prioritizes handling a large number of requests so that none of them have to wait too long to handle a single request in a minimum amount of time. This means that some requests will need to wait for a new Lambda instance to start, and some will not. The latency of processing a single request isn’t really deterministic.

Cold Start

A *cold start*, in the serverless jargon, is when an incoming request needs to wait for a new Lambda instance for processing. Early on, Lambda cold starts were a few seconds long on average. Many blog posts appeared on the topic of keeping some Lambdas in reserve, *warm*, to avoid cold starts. Since the early days, Lambda start-up times have improved significantly, so for most cases you can probably ignore that old advice. AWS does not publish any official information about cold starts, but empirical tests suggest that with JavaScript or Python, the cold start is less than half a second. With Java and C#, it still may take a bit longer depending on the application size, which is another reason why lightweight environments are preferred for Lambda. For a more reason why lightweight environments are preferred for Lambda. For a more

detailed (but still unofficial) analysis of cold start times, check out Mikhail Shilkov's analysis of Cold Starts

Virtual Private Cloud (VPC)

In late 2019 (after the first version of this book was published), AWS significantly improved start-up times for Lambda functions connected to virtual private clouds (VPC). For applications that cannot avoid a lengthy initialisation, they also enabled users to reserve minimum capacity, reducing the problem of cold starts.

Execution time limited to 15 minutes

Another major technical constraint is the total allowed execution time. Currently, a Lambda function is allowed to run for three seconds by default, and you can configure it to allow up to 15 minutes. If a task does not complete in the allowed time, Lambda will kill the virtual machine and send back a timeout error.

The 15-minute limit is a hard constraint now, so unless you have a very special relationship with AWS, you can't ask for a longer allowance. Long-running tasks need to be split and executed in different batches or executed on a different service. AWS offers some alternative services such as Fargate that cost more and start more slowly but can run for a longer period of time.

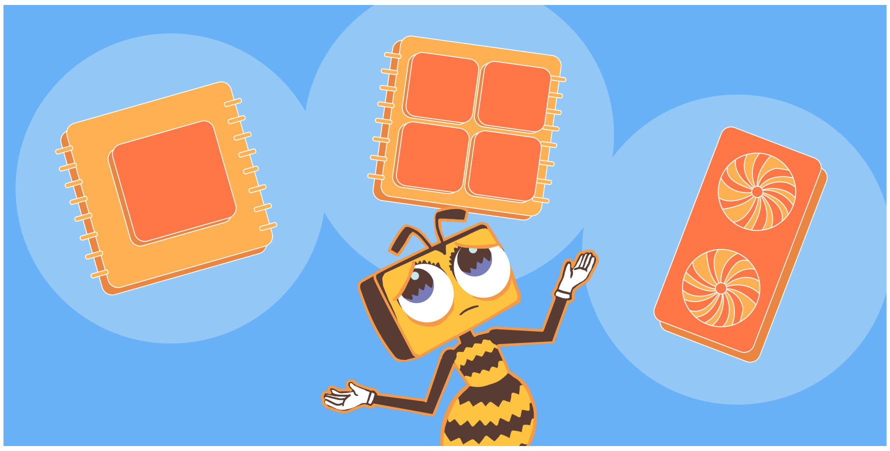
AWS Step Functions

In many cases, designing an application with Lambda in mind will help you work around the execution time constraint. For example, instead of using Lambda to start a remote

task and then waiting for it to complete, split that into two Lambdas. The first just sends a request to a remote service, and then the incoming response kicks off a different Lambda. You can use **AWS Step Functions** to coordinate workflows that last up to one year, and invoke Lambda functions when required. Don’t pay for waiting.

No direct control over processing power

The last major technical constraint of Lambda is around choosing processors. Today, it is usual for container execution environments to offer a whole buffet of processor combinations, including GPUs, various CPU speeds or numbers of cores, and instances optimised for certain types of tasks. However, with Lambda you don’t get to choose any of that. The only container choice you can make is the amount of memory, from 128 MB to about 3 GB. Lambda is not good for tasks that require GPUs.



Lambda CPU allocation

Memory configuration has an indirect impact on processing power. Lambda allocates CPU power in proportion to memory, so that at 1792 MB, a function has access to one full virtual CPU. With Node.js, all tasks run through a single core anyway, so with JavaScript you won’t get any further processing speed improvements if you ask for more than 1.75 GB. With Java or other languages that can take advantage of multiple cores, asking for the maximum allowed

memory might give you faster responses and lower cost for CPU-intensive tasks.

Lambda pricing depends on two factors: *basic price for memory allocation* and *time spent executing*. The fact that higher memory also leads to more CPU power can result in counter-intuitive cost structures. Allocating more memory raises the basic price, but more CPU power can lead to significantly shorter execution, reducing the price.

Because there are no direct controls for processing power, the best way to optimise costs and performance is to explore various parameter combinations. Luckily, AWS Lambda makes it easy to change these settings quickly and operate multiple versions in parallel cheaply. For tasks that become expensive or slow, I strongly suggest trying out several memory allocation options to find a sweet spot between price and performance. Alex Casalboni’s open source project AWS Lambda Power Tuning can help you visualize the performance for various configurations of your Lambda functions.

At this point, you will have become familiar with the financial and technical aspects of AWS Lambda. In the next lesson, you’ll discuss when it is a good option to use AWS Lambda.