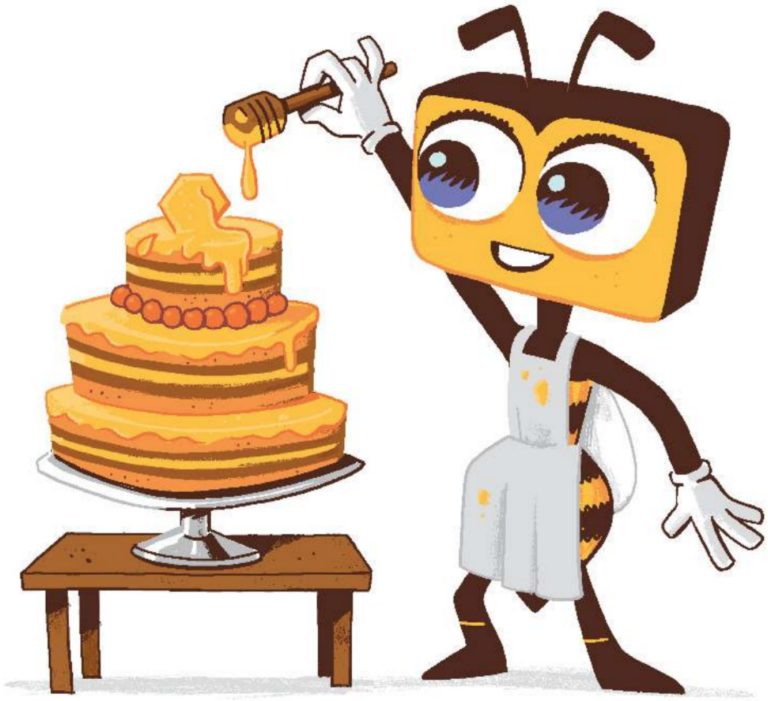
Deployment Options

In this lesson, you'll address some of the common misconceptions about Lambda.

WE'LL COVER THE FOLLOWING

In the final chapter of this course, you'll learn about common ways of structuring applications with Lambda functions.



Serverless architectures are still a relatively novel way of deploying applications, and it's too early to talk about generally applicable design patterns or best practices. The community is still discovering how best to use this type of deployment architecture, and the platform is still frequently changing. There are, however, some good design practices worth considering when you are thinking about structuring applications with Lambda functions.

As closing remarks for the course, I'd like to offer some ideas that helped us build and maintain MindMup efficiently. Hopefully, they will help you start thinking about organising your applications around Lambda functions in an effective way.

Think about jobs, not functions

AWS Lambda quietly appeared on the scene in 2014, several years before 'serverless' became a buzzword. Back then, the DevOps community was trying to figure out the right relationship between applications, virtual machines and containers. Tool vendors were competing on how many times they could cram 'micro-services' into their ads. 'Monolith' became a dirty word. AWS product managers decided to introduce a new term and called their deployment units 'functions'. It's possible that they did this on purpose to signal a generational shift in deployment architecture. To avoid comparisons with older products, they chose a name that did not come with a lot of existing connotations, at least not in the deployment area. However, the name Lambda has a very specific meaning in functional programming, which itself is based on a branch of mathematics called *Lambda calculus*. Mention Lambdas and functions together, and people with any experience of functional programming immediately make wrong assumptions about the AWS Lambda platform, such as that it is stateless, or that it can parallelise calculations automatically. That's why you need to unpick a terminological mess in order to discuss design and architecture of applications based on cloud functions.

Misconceptions about Lambda functions

People influenced by functional programming terminology often think that a Lambda function needs to be a small isolated piece of code with a single purpose (single processing function) and that it is stateless (or, in functional terminology, that has no side-effects). None of those assumptions are true.

You've mostly seen very simple functions as examples in this course, not because Lambda functions have to be simple, but because complex code isn't good for teaching. A single Lambda function can have a deployment package up to 250 MB, more than enough for an enterprise application server along with all the entity and session beans a UML architect can draw in a lifetime. There's nothing preventing you from bundling a big monolithic website and deploying it as a single Lambda function. Likewise, there's nothing preventing

you from deploying an API where each individual endpoint is handled by a separate Lambda function, served by hundreds of other Lambda functions that perform background tasks. And, of course, anything between those two extremes is also possible.

AWS Lambda function invocations are not technically independent, and assuming that they are can cause subtle problems. Lambda will often reuse existing container instances over a short period of time, to speed up processing and avoid cold starts. The initialisation code for the functions runs only once on start, not for each request, and the process in a Lambda instance keeps running to handle subsequent requests. For JavaScript, this means that any variables or modules you load outside the Lambda event handler stay around between two different requests. Leaving a mess in the internal memory state during one request can unexpectedly hurt when the next request comes in.

Lambda containers also have a temporary disk space (512 MB available under `/tmp`), and two different requests executing on the same function instance can, in theory, share data by reading and writing files there. If your code writes to the temporary disk space and does not clean up after each request, the function instance will run out of available space at some point.

People influenced by the container and DevOps terminology often think about Lambda functions as services, but that comparison also has problems. Traditional container-based services are usually stateful and long-running, with the application developers controlling routing. Lambda fully controls routing, starting and shutting down containers, and application developers don't have any control over it. Lambda functions aren't really stateless or stateful, they are transient.

Another common misconception about Lambda deployments is that it's impossible to control data ownership across instances. Lambda is an auto-scaling platform, but this doesn't mean that it has to scale. As an extreme example, it's quite easy to create a proper monolith. In Chapter 12, you saw how to control function concurrency limits in order to prevent abuse from runaway functions and denial-of-service attacks. By setting the concurrency limit of a function to 1, you can ensure that AWS never runs more than one instance of that code in parallel. Lambda will create the function container when it's needed, and shut it down when there's no more demand. For an

occasional task, using single-instance Lambda functions might be cheaper and easier to manage than similar code running within an AWS Elastic Compute Cloud (EC2) container. For a sustained load over a longer period of time, this would be several times more expensive than an EC2 machine, but, given that it's a single instance, it would not really break the bank.

The Bright Side

On the other hand, the fact that a Lambda function can have side-effects opens up some very nice opportunities. For example, the total size for a Lambda function package needs to be under 250 MB, including all layers. One way to work around that limit is to put the additional code on S3, compressed into an archive, instead of a Lambda layer. The Lambda function can then be just a small skeleton initialiser which downloads and unpacks the additional resources from S3 to the temporary disk space when it starts. Because the temporary space is shared between function executions, this costly initialisation can happen only once per cold start, and most requests will not suffer from the additional overhead.

Instead of thinking about deploying functions or services, it can be more useful to think about structuring Lambda functions around discrete jobs that don't necessarily perform a single code function. Instead of thinking about stateless or stateful services, it's more useful to design for share-nothing architecture (when different parts can own data and share state) but different functions and instances of a single function don't actively share data between themselves.

You're ready to move to the discussion in the next lesson!