One Lambda or Many?

Take a look at the constraints of Lambda functions in this lesson.

The common wisdom today seems to be that monolithic code is bad and that micro-services are good, but there is cause to disagree with this. Monoliths are usually simpler to develop and manage than a network of services, and in many cases significantly faster to deploy. Dividing code and data into multiple machines always introduces a huge amount of complexity related to synchronisation and consistency, which simply does not exist when everything is in a single memory space. Lambda provides many options for breaking down or aggregating processing, and it's important to look at the platform constraints so you can decide which is the best solution in a particular case.

Aggregate processing data ownership

The first important constraint of the Lambda platform is that application developers do not control request routing. Two logically related requests, for example coming from the same user or related to the same data entity, might be received by a single Lambda process or by two different processes running on two different instances. If a process is modifying or accessing data and it needs to ensure conceptual consistency, aggregate it into a single function (and ideally a single request) instead of splitting it into multiple functions.

Data consistency is much easier to achieve with a single process than with multiple processes, even if they run the same code. Martin Fowler is often misquoted online for his apparent First Law of Distributed Computing ('Don't'), but the actual law he wrote about in *The Patterns of Enterprise Architecture* is 'Don't distribute objects'. (For any domain-driven-design fans out there, this should really be 'Don't distribute aggregates'.) Having two different machines work on the same conceptual entity at the same time requires locking, transactions, and conflict resolution. When a single instance owns a conceptual data entity and doesn't need to coordinate with other instances running on different machines, all these problems are easily avoided.

If two pieces of data need to be consistent with each other, make sure they are processed by a single instance of a single function. In the most extreme case, limiting allowed concurrency to 1 ensures that only a single instance of a Lambda function ever runs, which can help avoid locking and conflict problems, but severely limits the total throughput of a process. Handling a lot of traffic often requires distribution, but the key to making this work is to avoid a network of tightly coupled systems. (The derogatory name for such architectures is a *distributed monolith*.) By aggregating code around data ownership and letting different instances own different blocks of data, you can avoid tight coupling between instances and benefit from Lambda's auto-scaling capabilities.

Note that a common argument in favour of distributed systems over single-instance services is that more machines typically have better resilience than a monolith, but with Lambda this does not apply. The platform will handle failover and recovery automatically even if you set the concurrency limit to 1.

If you need to guarantee that a data entity will never be modified by more than one instance of a Lambda function at a time, you don't have to restrict Lambda to a single instance. Put a Kinesis Data Stream between the requests and Lambda functions. Kinesis splits incoming events into shards (parallel processing units) and allows you to configure the partitioning data key. When Lambda reads events from a Kinesis stream, a single instance connects to a single shard and processes events in sequence. Although this doesn't make it possible to completely control routing between events and Lambda instances, it does make it possible to guarantee that there will be no more than one instance actively processing events for a specific key.

instance actively processing events for a specific key.

For example, at MindMup we use this trick to allow users to concurrently edit files. You can use the file identifier as the partitioning key for Kinesis shards. Regardless of how many users send concurrent requests, all events for a single file will always end up in the same shard (events for other files may go to different shards). Because only one Lambda instance reads from a single shard, it can apply changes to an external file without having to worry about locking or concurrent modifications. You can control the total throughput of the system easily by configuring the number of shards.

If you need to guarantee a particular order of processing messages, but do not care about limiting overall parallelisation, you can set up an SQS FIFO (First-in-first-out) Queue and connect it to a Lambda function. When Lambda pulls messages with associated group identifiers from SQS, it will send only one batch per group identifier in parallel. It will then wait for an instance to successfully process that batch before pulling more messages from the same group. For more information, check out the Using with Amazon SQS page from the AWS Lambda Developer Guide.

Aggregate code that needs to be consistent

The second important technical constraint of the Lambda platform is that different functions don’t deploy atomically. CloudFormation makes updates atomic in the sense that it will either completely succeed or completely roll back all the resources in a template, but it does not update all the Lambda functions at the exact same millisecond. It’s best to design the code assuming that there will be a short period of time, even if it’s a few milliseconds, where different functions may run with different versions of application code (or even that there will be two different versions of the same function running in parallel). As a consequence, if two pieces of code need to be fully consistent with each other at all times, put them in the same function. Code consistency is much easier to achieve with a single function than with multiple functions.

If several Lambda functions share data through an external repository, such as an S3 bucket or an SNS topic, consider adding a format version to the data when it leaves the Lambda function. For example, add a format version to the file metadata when saving a file to S3, or put the format version into message attributes when sending a message to an SNS topic. Format versions make it possible to reduce the need for strong code consistency across different

functions. Data consumers can know what version of code saved or sent a particular piece of information, so they can decide to upgrade it, ignore some fields, or fall back to some alternative processing method for older events. This is a great trick for dealing with potential short inconsistencies between active versions in a large application.

Divide code around security boundaries

The third important technical constraint is that IAM security roles apply to whole functions. Different functions can have different security policies, but different parts of a function always run under the same privileges. If you need to reduce or increase the security privileges for some part of a process, it's better to extract that part into a separate Lambda function with a separate IAM role. Technically, you could use the AWS Security Token Service to change the security context inside a Lambda function, but doing that will be significantly more complicated than extracting a new function.

Smaller, more focused functions are less risky from a security perspective than a single bigger function that performs many tasks. When a Lambda function handles multiple types of requests, it will need to have a sum of all the required security privileges. Because it is difficult to audit and tightly evaluate a complex set of tasks, aggregated services usually get full access rights on back-end resources. For example, a typical application server gets full read-write access to a user database. A single security hole in a third-party dependency of a monolith can let attackers steal a lot of data. When each task runs as an individual Lambda function, it's much easier to give it the minimum required access level. Breaking code down into smaller parts also makes it easier to audit and reduce security levels as a system evolves.

When deciding whether two pieces of code should stay together, consider whether they need the same access levels to other resources. If one function needs to modify some security-sensitive user data, and another function just needs to read that information, you can improve overall security by isolating the code into two different functions and giving the functions different IAM roles.

Divide code around CPU and memory needs

The fourth important constraint is financial instead of technical. Because Lambda charges for memory allocation multiplied by CPU time, bundling tasks with different memory needs can be unnecessarily expensive.

For example, at MindMup we run exporters for various formats. The PDF and SVG exporters share about 99% of the code. The PDF actually first produces an SVG image then runs an external tool to convert it to PDF. The external tool requires a lot of memory. The SVG exporter itself just works with a text file stream and needs very little memory. Putting both exporters into the same Lambda function would require us to pay for the memory needs of the PDF process when users export SVG files. Separating them allows us to reduce the allowed memory for the SVG function and pay significantly less.

People often bundle related code together just because the process is the same. When two different usages of the same piece of code have different CPU or memory needs, isolating those usages into two different functions can save a lot of money. Think about jobs (use cases) when deciding on Lambda granularity, not code units (functions).

Divide tasks around timing restrictions

A single Lambda invocation is limited to 15 minutes. For tasks that take longer, Lambda may not be the appropriate solution, or you may be able to work around the limit by splitting a task into several steps and potentially parallelising the work.

For example, I worked on an application to process documents uploaded by users. Processing each page took about 30 seconds, so Lambda functions could easily deal with relatively short documents, but users could upload documents with hundreds of pages. Instead of trying to do everything in a single task, we broke it down into three functions. The first would just split the uploaded document into pages and upload individual pages to another S3 bucket. It could easily process hundreds of pages in a few minutes. The second function would trigger when a page was uploaded to S3, and always process a single page. We could run hundreds of these functions in parallel. The third function would combine the results when the conversion process was completed. Dividing the task into three functions like that made it possible to run everything in Lambda, but it also produced much faster results for users.

Dividing a job into smaller tasks due to timing restrictions typically requires an additional process to coordinate the execution of subtasks, periodically check for results, and report on overall errors or success. One option for such an umbrella process is to move it to a client device, similar to the solution in Chapter 11. Another option is to implement the coordination inside AWS, but doing it as a Lambda function may not be the best idea, because the controller would still be restricted by the 15 minutes time limit. Instead, consider using AWS Step Functions. Step functions are a mechanism for automating workflows with AWS resources, taking up to a year. You can configure parts of the workflow to invoke Lambda functions, pause or loop, and pass work between Lambda and other AWS services.

For tasks that cannot be parallelised, consider using AWS Fargate. Fargate is a task management system for the Amazon Elastic Container Service with similar billing to Lambda functions. It will start and scale containers automatically based on demand, and only charge you for actual usage. There is no specific duration limit for Fargate tasks, so a single job can run for as long as it needs.

Fargate tasks start significantly more slowly than Lambda functions, but if the expected duration for a task is longer than 15 minutes, a few dozen seconds of warm-up time won't really make a noticeable difference. Instead of function source code, you'll need a Docker container image to deploy tasks to Fargate, but on the other hand, that approach is a lot more flexible for packaging operating system utilities and third-party software. A single task can use up to 10 GB for the container storage, making Fargate a good option for situations where the 250 MB size limit for a function is too restrictive.

In the next lesson, you'll learn about the sharing behavior of Lambda functions.