SERVERLESS Architectures ON AWS

SECOND EDITION

Peter Sbarski
Yan Cui
Ajay Nair

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Serverless Architectures on AWS
Second Edition
Version 6

For more information on this and other Manning titles go to
manning.com

©Manning Publications Co. To comment go to liveBook

welcome

Dear Reader,

Firstly, thank you for reading the Second Edition of Serverless Architectures on AWS. We are delighted that you are interested in learning about Serverless architectures and have chosen our book to help you on your journey.

Since our last MEAP update there have been a lot of improvements and advances in AWS and Serverless technologies. Several exciting features launched for AWS Lambda and API Gateway. The whole platform matured, and more patterns and best practices emerged (a lot of them spearheaded by Yan Cui). This forced us to re-think our book and delay the publication of some of our chapters. We thought long and hard and decided to introduce case studies and cover technologies such as GraphQL, which weren't in our plan before. Our changes, unfortunately, delayed the publication of new chapters.

Today we are thrilled to announce that we are back with heaps of new reading material for you! We have two brand new chapters hot off the press. These two chapters describe real-world use-cases we experienced ourselves. One is about a Serverless production system built on REST, while the other covers a large platform built with GraphQL.

As always, we are keen to hear your feedback. Please let us know on the forum what you like or dislike. What would you like to see more (or less) from us? How can we make this the best book on architecture, serverless, and cloud-centric design for you?

Thank you again, and we hope you enjoy the two new chapters.

With regards,
Yan, Ajay, and Pete

P.S. There's even more great stuff in the pipeline for you.

brief contents

1

Going serverless

This chapter covers

- Traditional system and application architectures
- Key characteristics of serverless architectures and their benefits
- How serverless architectures and microservices fit into the picture
- Considerations when transitioning from server to serverless
- What's new in this second edition?

If you ask software developers what software architecture is, you might get answers ranging from "it's a blueprint or a plan" to "a conceptual model" to "the big picture." This book is about an emerging architectural approach that has been adopted by developers and companies around the world to build their modern applications – *serverless architectures*. Serverless architectures have been described as somewhat of a "nirvana" for an application architectural approach, in that it promises developers the ability to iterate as fast as possible while maintaining business critical latency, availability, security, and performance guarantees – with minimal effort on the developers' part. This book shows how to build these next generation of systems that can scale and handle demanding computational requirements without having to provision or manage a single server. Importantly, this book describes techniques that can help developers quickly deliver products to market while maintaining a high level of quality and performance by using services and architectures offered by today's cloud platforms.

1.1 What's in a name?

Before going in any further, we think it's important to get grounded on the term "serverless." There are various attempts at this already (including an official one from AWS https://aws.amazon.com/serverless/ and a community favorite from Martin Fowler https://martinfowler.com/articles/serverless.html). Here's how we define it:

TIP Serverless is a qualifier that can be applied to any software, which requires both that the software is consumed a utility service and incurs cost only when used.

Simple enough, right? But there's a lot to unpack in that simple definition. Let's dive into each of the two required criteria to call something serverless:

- Consumed as a utility service: The "software as a service" consumption model is well understood. It means that anyone using the software uses a prescribed application programming interface (API) or web interface to use the software and customize it, while staying within any published constraints for the software and usage policies for the API. Salesforce, Office365, and Google Maps are well known software package delivered as a service. What's key here is the fact that the actual infrastructure (servers, networking, storage etc.) hosting the software and powering the API is completely abstracted from you as the consumer – all that is visible (and all that matter) is what the API permits. A service also typically comes with accompanying availability, reliability, and performance guarantees from the service provider. A *utility* service further has the billing characteristics we expect from any utility computing offering, that is, you pay for usage, not for reservation, subscriptions, or provisioning. All existing public cloud offerings have some form of utility billing associated with them. For example, Amazon Elastic Compute (EC2) allows you to pay by the second for renting virtual machines the Amazon Web Services (AWS) cloud.
- Incurs cost only when used: This means there's zero cost for having the software deployed and ready to use. Think of this as the same cost model we expect from our public utilities like Electricity and Water – you as the consumer pay a per granular usage unit cost if we use any, but you pay zero if you use nothing. This aspect of pure usage-based pricing is a distinguishing criterion of serverless offerings from the other utility services that came before it.

Just to clear up any misperceptions…

One of the common misunderstandings is that the "-less" in "serverless" implies "absence of or without" (think "sugarless," "boneless," and so on), which leads to some colorful debates on social media on how any application architecture can claim to run without servers. We think of "-less" here means "invisible in context of usage" (think "wireless," "tasteless"). There obviously are servers somewhere! The difference is that these servers are hidden from you. There's no infrastructure for you to think about and no way to tweak the underlying operating system or virtual hardware configuration. Someone else takes care of the nitty-gritty details of infrastructure management, freeing your time for other things.

In the rest of the book, we will use the "serverless" qualifier only for software that fits these criteria. For example, Software that requires you to provide servers to host the software (like Apache web server) or configure the servers running the software would not qualify since it does not meet the first criteria. Software that is available as a service but requires to you to pay by subscription (like Salesforce) would not qualify since it does not meet the second criteria.

A serverless architecture, by extension, is one composed entirely of serverless components. But which components of an architecture need to be serverless for it to be called as such? Let's look at this next with an example.

1.2 Understanding serverless architectures

Let's take the example of a typical data-driven web application, not unlike the systems powering most of today's web-enabled software. These consist of a backend that accepts HTTP requests from the front end and then processes the requests. The backend servers perform various forms of computation and the client-side front ends provide an interface for users to operate via their browser, mobile, or desktop device. Data might travel through numerous application layers before being saved to a database. The backend, finally, generates a response—it could be in the form of JSON or fully rendered markup—which is sent back to the client (figure 1.1). These kinds of applications are conventionally architected as "tiers" – a presentation tier that controls how the information is captured and provided to the user, an application tier that controls the business logic of the application, and a data tier with the database and corresponding access controls.

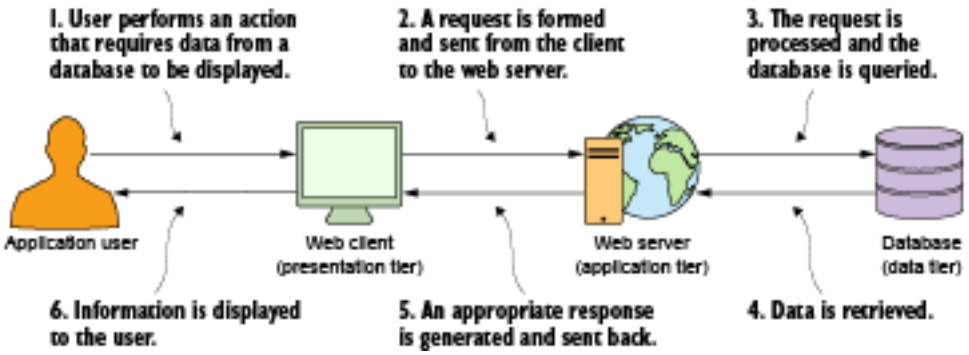


Figure 1.1 This is a basic request-response (client-server) message exchange pattern that most developers are familiar with. There's only one web server and one database in this figure. Most systems are much more complex.

Software architectures has evolved from the days of code running on a mainframe to multitier architecture where the presentation, data, and application/logic tiers are traditionally separated. Within each tier there may be multiple logical *layers* that deal with particular aspects of functionality or domain. There are also cross-cutting components, such as logging or exception-handling systems, that can span numerous layers. The preference for layering is understandable. Layering allows developers to decouple concerns and have more maintainable applications. Figure 1.2 shows an example of a tiered architecture with multiple layers including the API, the business logic, the user authentication component, and the database.

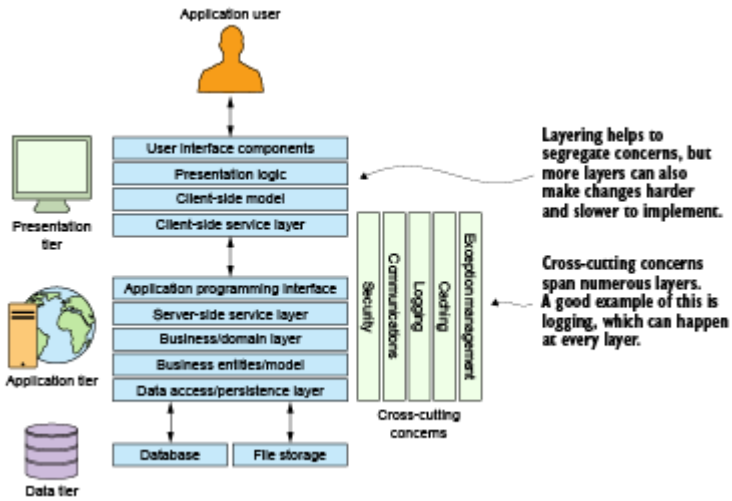


Figure 1.2 A typical three-tier application is usually made up of presentation, application, and data tiers**Error! Bookmark not defined.**. A tier may have multiple layers with specific responsibilities.

Tiers vs. layers

There is confusion among some developers about the difference between layers and tiers. A *tier* is a module boundary that exists to provide isolation between major components of a system. A presentation tier that's visible to the user is separate from the application tier, which encompasses business logic. In turn, the data tier is another separate system that can manage, persist, and provide access to data. Components grouped in a tier can physically reside on different infrastructures.

Layers are logical slices that carry out specific responsibilities in an application. Each tier can have multiple layers within it that are responsible for different elements of functionality such as domain services.

Now consider how you would architect this application today.

1.2.1 Service-oriented architecture and microservices

One blunt approach would be to combine all the layers – the API, the business logic, the user authentication – into one single, monolithic code base. This may sound like an anti-pattern today, but that was indeed the approach we adopted in the early days of cloud-based development. Most modern approaches dictate that you architect with reusability, autonomy, composability, and discoverability in mind. Among system and application architectures, service-oriented architectures (SOA) has a lot of name recognition among software developers, since it doesn't dictate the use of any particular technology. Instead, it encourages an architectural approach in which developers create autonomous services that communicate via message passing and often have a schema or a contract that defines how messages are created or exchanged.

The modern incarnation of the service-oriented approach is often referred to as microservices architectures. Modern application architectures are composed or services, communicating through events and APIs, with business logic inserted as appropriate. You should think of microservices as small, standalone, fully independent services built around a particular business purpose or capability. Ideally, microservices should be easy to replace, with each service written in an appropriate framework and language. The mere fact that microservices can be written in different general-purpose or domain-specific languages (DSL) is a drawing card for many developers. Benefits can be gained from using the right language or a specialized set of libraries for the job. Each microservice can maintain state and store data. And if microservices are correctly decoupled, development teams can work and deploy microservices independently of one another. This approach of building and deploying applications as a collection of loosely coupled services is considered the default approach to development in the cloud today (the "cloud native" approach, if you will).

Microservices all the time?

Microservice approaches aren't all a bed of roses. Having a mix of languages and frameworks can be hard to support, and, without strict discipline, can lead to confusion down the road. Eventual consistency, transaction management, and complex error recovery can make things more difficult.

Coming back to our example web application, you can imagine building it with the combination of a few microservices, each representing a layer – one each for authentication, reporting, database access, and the primary one representing the business logic, each with its own API. The presentation layer would then be responsible for sequencing the calls across the different services.

1.2.2 Implementing the architecture the conventional way

Once you have decided how your application is going to be architected, and all the software required for each of the layers is ready to go, you would think the hardest part is done. The truth is, that's when some of the more complex tasks begin. Developing your desired services traditionally requires servers running in data centers or in the cloud that need to be managed, maintained, patched, and backed up. Today, you would pick from a few options -

- **Directly build on VMs** - The physical deployment of each service requires you to have a set of instances, with additional tasks to address required activities such as load balancing, transactions, clustering, caching, messaging, and data redundancy. Provisioning, managing, and patching of these servers is a time-consuming task that often requires dedicated operations people. A non-trivial environment is hard to set up and operate effectively. Infrastructure and hardware are necessary components of any IT system, but they're often also a distraction from what should be the core focus—solving the business problem. In our simple web application example, you would have to become an expert in building distributed systems and cloud infrastructure management.
- **Use a PaaS** – Over the past few years, technologies such as platform as a service

(PaaS) and containers have appeared as potential solutions to the headache of inconsistent infrastructure environments, conflicts, and server management overheard. PaaS is a form of cloud computing that provides a platform for users to run their software while hiding some of the underlying infrastructure. To make effective use of PaaS, developers need to write software that targets the features and capabilities of the platform. Moving a legacy application designed to run on a standalone server to a PaaS service often leads to additional development effort because of the ephemeral nature of most PaaS implementations. Still, given a choice, many developers would understandably choose to use PaaS rather than more traditional, more manual solutions thanks to reduced maintenance and platform support requirements.

- **Use containers** - Containerization is considered ideal for microservices architectures since it is a way of isolating an application with its own environment. It's a lightweight alternative to full-blown virtualization that traditional cloud servers use. They're an excellent deployment and packaging solution when dependencies are in play, albeit with their own housekeeping challenges and complexities. Containers are isolated and lightweight but they need to be deployed to a server—whether in a public or private cloud or onsite.

While each of these models are perfectly valid, and offer varying degrees of simplicity and speed of development for your service, your costs are still driven by the lifecycle of the infrastructure or servers you own – not to your application usage. If you purchase a rack at the data center, you pay for it 24/7. If you purchase a cloud instance (wrapped in a PaaS or running containers or otherwise) you pay for it whenever it is running, independent of whether it is serving traffic for your web app or not. This leads to an entire discipline of engineers investing in improving server efficiency, or trying to match infrastructure lifecycle to application usage, and trying to match server sizes to traffic patterns. This also means all the effort spent on these tasks is time taken away from improving the functionality and differentiating aspects of your application. This is equivalent to asking for a place to plug in your appliance, and being asked to pay for a share of the power generators at your utility company, and also configuring the generator to deliver the power in the phase, frequency, and wattage you desire – no matter how much you use. The actual outcome (plug in your appliance) is dwarfed by the effort and cost for the infrastructure required (the generators). This is where the serverless approach comes in. It aims for the moral equivalent of the utility approach we know and love today – there when you need it, complexity abstracted away, and only pay for when you use it.

1.2.3 Implementing architecture the serverless way

A serverless architecture of our sample application would be composed of its layers (in figure 1.3) implemented using a serverless approach. For example, to build the API, we would leverage a service to build the API that does not cost us anything if there are no API calls. To build the authentication service, we would leverage a service to build the authentication capability that does not cost us anything if there are no authentication calls. To build the storage service, we would leverage…you get the picture. Much like a public cloud approach

offered virtual infrastructure Legos to assemble your cloud stack in the early days, a serverless architecture leverages existing services from cloud providers like AWS to implement its architectural components. As an example, AWS offers services to build your application primitives like APIs (Amazon API Gateway), workflows (AWS Step Functions), queues (Amazon Simple Queue Service), databases (DynamoDB and Aurora) and more.

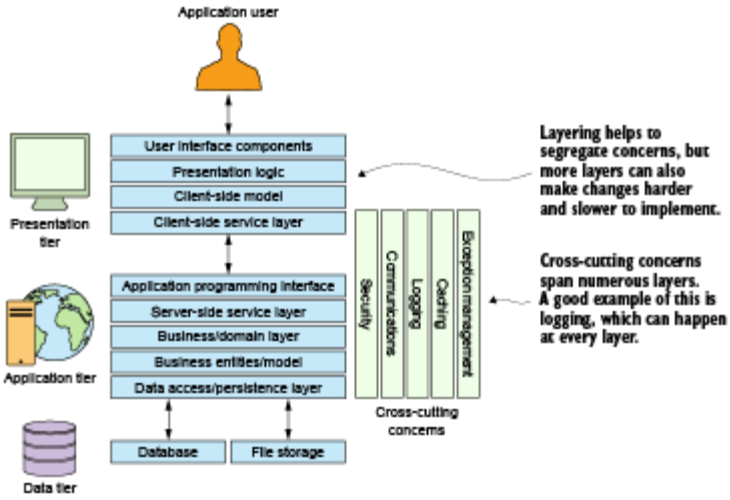


Figure 1.3 A typical three-tier application is usually made up of presentation, application, and data tiers. A tier may have multiple layers with specific responsibilities, and each layer can be implemented with a serverless approach.

The idea of leveraging off-the-shelf services to implement parts of your architecture is not new – indeed, it's been a best practice since the days of SOA. What's changed in the last few years is the new capability to also implement the *custom* aspects of your application (like the business logic) in a serverless manner. This ability to run arbitrary code without having the provision infrastructure (to run it as a service) or to pay for the infrastructure is referred to as *Functions as a Service* (FaaS), and is central to serverless architectures. A FaaS allows you to provide custom code and associated dependencies (written in any language of your preference) and some configuration to dictate your desired performance and access control characteristics. The FaaS then executes this unit, referred to as a *function*, on an invisible compute fleet, with each execution of your code receiving an isolated environment with its own disk, memory, and CPU allocation, and you pay only for the time your code runs. A function is not a lightweight instance – instead, think of it akin to processes in an OS, where you can spawn as many as needed by your application, and then spin them down when your application isn't running.

Figure 1.4 shows a sample implementation of the web application we've been discussing using a serverless approach. All of this can be done in an organized way to prevent spaghetti implementations and dependency nightmares by clearly defining service boundaries, allowing functions to be autonomous, and planning how functions and the services implementing other layers will interact.

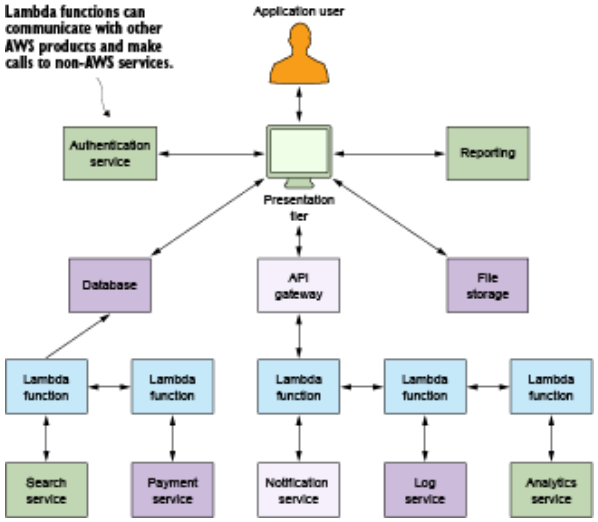


Figure 1.4 In a serverless architecture there's no single traditional back end. The front end of the application communicates directly with services, the database, or compute functions via an API gateway. Some services, however, must be hidden behind compute service functions, where additional security measures and validation can take place. [TODO: NEEDS NEW DIAGRAM]

More on FaaS

AWS's FaaS offering is called AWS Lambda, and is one of the first from the major cloud providers. Chapter 6 covers Lambda in more detail, including its methods of invocation, configuration, and best practice with regard to design. Note that Lambda isn't the only game in town. Microsoft Azure Functions (http://bit.ly/2DWx5Gn), IBM Cloud Functions (http://bit.ly/2l1PWbd), and Google Cloud Functions (http://bit.ly/2CbzOem) are other FaaS services you might want to look at.

Many developers have conflated serverless with FaaS offerings AWS Lambda (and still do), which often leads to confusing statements like adopting containers vs. serverless when they really mean containers or functions. We like how TJ Hallowaychuk, the creator of the popular Apex framework, defines what serverless is about. He once tweeted, "serverless != functions, FaaS == functions, serverless == on-demand scaling and pricing characteristics (not limited to functions) ." We couldn't agree more.

An emerging trend is that of "serverless containers" i.e. leveraging containers instead of functions to implement the custom logic, and using the container as a utility service and incurring costs only when the container runs. Services like AWS Fargate offer this capability. The difference between the two (functions vs. containers) is just the degree to which customers want to shift the boundaries of shared responsibilities. Containers give you a bit more control over user space libraries and network capabilities. Containers are an evolution of the existing server based/VM model, offering an easy packaging and deployment model for your application stack. You still are required to define your OS requirements, your desired language stack, and dependencies to deploy code, which means you continue to carry some of the infrastructure complexity. For the purpose of this book, we are going to focus on leveraging FaaS for our custom logic, though you can explore the usage of serverless containers for the same as well.

Serverless architectures is really the culmination of shifts that has been going for a long time – from tiers to granular layers, from monoliths to services, and from managing infrastructure to increasingly delegating the undifferentiating responsibilities. Because each layers is a service, each of the interactions between the different components of the architecture happens via a secure API call, and each layer can independently scale, be reliable, and performant– all without incurring any cost when idle, and without having to manage any infrastructure. Since the serverless service only costs you for the API call and associated action, you only pay when your application is doing useful work. Serverless architectures can also help with the problem of layering and having to update too many things. There's room for developers to remove or minimize layering by breaking the system into functions and allowing the front end to securely communicate with services and even the database directly. A well-planned serverless architecture can make future changes easier, which is an important factor for any long-term application.

To recap, a serverless architecture leverages a serverless implementation for each of its components, leveraging FaaS like AWS Lambda for custom logic. This means each component is built as a service with utility, pay-per-use pricing and incurs cost only when used. Each component is a service and exposes no configuration or cost related to the infrastructure it is running on, which means these architectures don't rely on direct access to a server to work. By making use of various powerful single-purpose APIs and web services, developers can build loosely coupled, scalable, and efficient architectures quickly. *Moving away from servers and infrastructure concerns, as well as allowing the developer to primarily focus on code, is the ultimate goal behind serverless.*

1.3 Making the call to go serverless

The web application example we just went through is probably the simplest example of what you can achieve with serverless architectures. A serverless approach can work exceptionally well for companies that want to innovate and move quickly. Functions and serverless architectures in general are extremely versatile. You can use them to build back ends for CRUD applications, e-commerce, back-office systems, complex web apps, and all kinds of mobile and desktop software. Tasks that used to take weeks can be done in days or hours as long as the right combination of technologies is chosen. Lambda functions are stateless and scalable, which makes them perfect for implementing any logic that benefit from parallel processing. The most flexible and powerful serverless designs are event-driven, which means each component in the architecture reacts to a state change or notification of some kind, rather than responding to a request or polling for information. In chapter 3, for example, you'll build an event-driven, push-based pipeline to see how quickly you can put together a system to encode video to different bitrates and formats.

> **NOTE** You will find the use of events as a communication mechanism between components to be a recurring theme in serverless architectures; indeed, AWS Lambda's initial launch was as an event driven computing service. Building event-driven, push-based systems will often reduce cost and complexity (you won't need to run extra code to poll for changes) and potentially make the overall user experience smoother. It goes without saying that although event-driven, push-based models are a good goal, they might not be appropriate or

achievable in all circumstances. We'll cover different event models and you'll work through examples in later chapters.

Serverless architecture allows developers to focus on software design and code rather than infrastructure. Scalability and high availability are easier to achieve, and the pricing is often fairer because you pay only for what you use. More importantly, you have a potential to reduce some of the complexity of the system by minimizing the number of layers and amount of code needed. Adopting a serverless approach to application development comes with significant agility, elasticity, and cost efficiency gains. However, it is easy to fall into the trap of trying to adopt a serverless approach for *all* applications. We recommend keeping a few principles in mind as you start your serverless journey.

- **Avoid lift-and-shift** - In practice, serverless architectures are more suited for net new applications than porting existing applications over. This is because existing application code bases have a lot of code that is made redundant by the serverless services. For example, porting a Java Spring app into Lambda brings a heavy framework into a function, most of which exists to interact with a web server (which doesn't exist inside Lambda).
- **Adopt a serverless first approach, not a serverless only approach -** While there are many companies (like A Cloud Guru) who have adopted a serverless only approach where 100% of their application runs as a serverless implementation, the more widespread approach that companies like Expedia and T-Mobile have adopted is to go serverless first. What this means is that their developers attempt to first build any new application in the following priority order – build as much using third party services, fall back to custom services built using AWS serverless primitives (like AWS Lambda), and finally fall back to custom services built using custom software running on infrastructure like EC2. We talk about reasons why you may have to fall back beyond custom serverless services in the next section.
- **It doesn't have to be all or nothing** - One advantage of the serverless approach is that existing applications can be gradually converted to serverless architecture. If a developer is faced with a monolithic code base, they can gradually tease it apart and convent individual components into a serverless implementation. The best approach is to initially create a prototype to test developer assumptions about how the system would function if it was going to be partly or fully serverless. Legacy systems tend to have interesting constraints that require creative solutions; and as with any architectural refactors at a large scale, there are inevitably going to be compromises. The system may end up being a hybrid—see figure 1.5—but it may be better to have some of its components use Lambda and third-party services rather than remain with an unchanged legacy architecture that no longer scales or that requires expensive infrastructure to run.

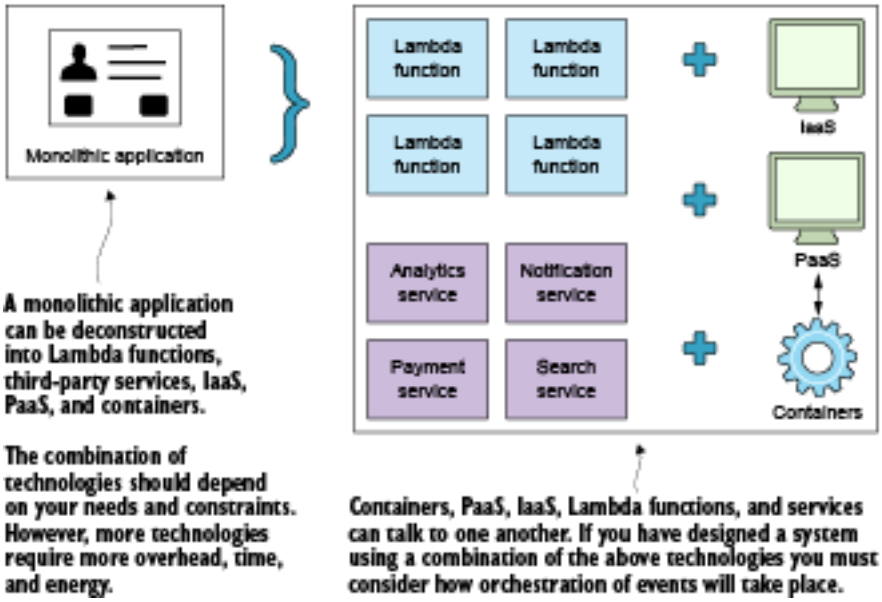


Figure 1.5 Serverless architecture is not an all-or-nothing proposition. If you currently have a monolithic application running on servers, you can begin to gradually extract components and run them in isolated services or compute functions. You can decouple a monolithic application into an assortment of infrastructure as a service (IaaS), PaaS, containers, functions, and third-party services if it helps.

The transition from a legacy, server-based application to a scalable serverless architecture may take time to get right. It needs to be approached carefully and slowly, and developers need to have a good test plan and a great DevOps strategy in place before they begin.

What about Ops?

Early on, around the time of the first conference on serverless technologies and architectures (serverlessconf.io), there was talk that serverless technologies foreshadowed the era of *NoOps*. Some people believed that thanks to serverless, companies would no longer need to think about infrastructure operations. The cloud vendor will take care of everything was the thought. That assumption, that NoOps was a real thing, proved not to be the case.

When it comes to building and running serverless applications, DevOps engineers are essential except now they have a different focus. Their attention is on deployment automation, application testing and deployment, and working with the operations/support teams of their preferred cloud provider (rather than tweaking servers and patching Operating Systems). Companies can get away with smaller, more specialized DevOps teams; however, ignoring operations entirely is a recipe for disaster (and don't let anyone else tell you otherwise). Remember, when your application fails, customers will hold you accountable, not your cloud provider, so be ready and have the right people and processes in place.

- **Pick applications suited for a service oriented architecture**: Serverless

architectures are a natural extension of ideas raised in SOA. In serverless architecture all custom code is written and executed as isolated, independent, and often granular functions that are run in a compute service such as AWS Lambda. Since every component is a service, serverless architectures share a lot of advantages and complexities with event driven microservices architectures. This also means applications likely need to be architected to meet the requirements of these approaches (like making the individual services stateless, for example). However, keep in mind that the serverless approach is all about reducing the amount of code you have to own and maintain, so you can iterate and innovate faster. This means you should strive to minimize the number of components that are required to build your application. For example, you may architect your web application with a rich front end (in lieu of a complex back end) that can talk to third-party services directly can be conducive to a better user experience. Fewer hops between online resources and reduced latency will result in a better perception of performance and usability of the application. In other words, you don't have to route everything through a FaaS - your front end may be able to communicate directly with a search provider, a database, or another useful API. However, keep in mind that moving from a monolithic approach to a more decentralized serverless approach doesn't automatically reduce the complexity of the underlying system. The distributed nature of the solution can introduce its own challenges because of the need to make remote rather than in-process calls and the need to handle failures and latency across a network, which your application will need to be resilient to.

- **Minimize custom code**: The rise of serverless means many standard application components like APIs, workflows, queues, and databases are available as serverless offerings from cloud providers and third-parties. It's far more useful for developers to spend time solving a problem unique to their domain than re-creating functionality already implemented by someone else. Don't build for the sake of building if viable third-party services and APIs are available. Stand on the shoulders of giants to reach new heights. Appendix A has a short list of Amazon Web Services and non-Amazon Web Services we've found useful. We'll look at most of those services in more detail as we move through the book. However, it goes without saying, however, that when a third-party service is considered, factors such as price, capability, availability, documentation, and support must be assessed. If you have to build a piece of custom functionality, our advice is simple: try to solve your problem using functions first, and if that doesn't work explore containers and more traditional server-based architectures second. Developers can write functions to carry out almost any common task, such as reading and writing to a data source, calling out to other functions, and performing a calculation. In more complex cases, developers can set up more elaborate pipelines and orchestrate invocations of multiple functions.

1.4 Serverless pros and cons

The serverless approach of building applications by quickly assembling services provides two significant advantages to you - less code to write and maintain per application, and per-activity pricing on their application. This translates into a disruptive gain in agility/developer productivity, and a much more streamlined alignment between development and finance (since

any application inefficiencies or optimizations show a direct, tangible financial impact). Here are a few of the specific benefits you will realize by adopting serverless architecture:

- **High scale and reliability without server management:** Building large scale distributed systems is hard. Tasks such as server configuration and management, patching, and maintenance are taken care of by the vendor, as is managing the infrastructure architecture for high scale and reliability, which saves time and money. For example, Amazon looks after the health of its fleet of servers that power AWS Lambda. If you don't have specific requirements to manage or modify server resources, then having Amazon or another vendor look after them is a great solution. You're responsible only for your own code, leaving operational and administrative tasks to a different set of capable hands.
- **Competitive Pricing:** Traditional server-based architecture requires servers that don't necessarily run at full capacity all of the time. Scaling, even with automated systems, involves a new server, which is often wasted until there's a temporary upsurge in traffic or new data. Serverless systems are much more granular with regard to scaling and are cost--effective, especially when peak loads are uneven or unexpected. Because of their utility, pay-per-use billing, serverless services can be extremely cost-effective; however, they're not cheaper than traditional (server and container) technologies in all circumstances. The best thing is to do a bit of modeling before embarking on a big project.
- **Less code:** We mentioned at the start of the chapter that serverless architecture provides an opportunity to reduce some of the complexity and code in comparison to more traditional systems. Adopting a serverless approach eliminates undifferentiated code, such as that required for orchestrating server fleets, code for routing requests and events between components which forms a surprisingly large part of modern code bases.

However, Serverless is not a silver bullet in all circumstances. Here are some reasons where you would want to avoid serverless architectures.

- **You are not comfortable with public cloud-based architectures –** Serverless development is a natural extension of the move cloud-based development, where more and more of the undifferentiated heavy lifting is moved to the providers. There are applications and business scenarios where you need to maintain your own data center; in such cases you cannot build a serverless architecture (though you are welcome to host your own primitives on your infrastructure and use those to build applications).
- **The services don't meet the availability, performance, compliance, or scale needs of your customers** - AWS serverless services offer an availability SLA, but their threshold maybe below what you need for your business. They also have a variety of compliance certifications, but you must validate if they need what your business need. Services like AWS Lambda also do not offer a performance SLA which means you may need to evaluate their performance against your desired levels. Non-AWS third-party services are in the same boat. Some may have strong SLAs, whereas others may not have one at all.
- **Your application and business needs more control or you need to customize**

the infrastructure - When it comes to Lambda, the efficiencies gained from having Amazon look after the platform and scale functions come at the expense of being able to customize the operating system or tweak the underlying instance. You can modify the amount of RAM allocated to a function and change timeouts, but that's about it (see chapter 6 for more information). Similarly, different third-party services will have varying levels of customization and flexibility.

- **Your application and business needs require you to stay vendor agnostic** - If a developer decides to use third-party APIs and services, including AWS, there's a chance that architecture could become strongly coupled to the platform being used. The implications of vendor lock-in and the risk of using third-party services—including company viability, data sovereignty and privacy, cost, support, documentation, and available feature set—need to be thoroughly considered.

In this chapter you learned what serverless architecture is, looked at its principles, and saw how it compares to traditional architectures. In the next chapter, we'll explore important architectures and patterns, and we'll discuss specific use cases where serverless architectures were used to solve a problem.

1.5 What's new in this second edition?

For all intents and purposes, this is a completely different book from the first edition of Serverless Architectures on AWS. Most of the chapters have been written from the ground up to provide a completely different experience to the first edition.

When the first edition of this book came out in 2017, serverless was still very new and many of you were learning about serverless for the first time. As such, the first edition gives you a gentle introduction to serverless and walks you through building a serverless application step-by-step. Since then, the adoption of serverless technologies has grown steadily and really accelerated in the last 12-18 months. At the same time, much more content has become available online, including numerous books and video courses that help you get started with serverless technologies on AWS.

If you're looking for an introduction to serverless on AWS, we have included some introductory chapters in appendices A and B to help you get started. You can also find the first edition of this book on the Manning website.[1] Most of the content from the first edition is still relevant today and you will learn to build a serverless application from scratch.

But, just as serverless technologies allow us to focus on the things that differentiate our business, we want to focus on things that can differentiate this book with this second edition. So instead of yet another getting started guide to serverless, this book focuses on serverless use cases and how to address architectural concerns such as scalability and cost. It is aimed at developers with some experience of serverless technologies already and answers the questions that many of you have been asking us. And given the switch in focus, this book does not have many actual code samples. Instead, we hope to challenge the way you think about architecture and help you get the most out of serverless technologies on AWS.

[1] https://www.manning.com/books/serverless-architectures-on-aws

1.6 Summary

- The cloud has been and continues to be a game changer for IT infrastructure and software development.
- Software developers need to think about the ways they can maximize use of cloud platforms to gain a competitive advantage.
- Serverless architectures are the latest advance for developers and organizations to think about, study, and adopt. This exciting shift in implementing application architectures will grow quickly as software developers embrace compute services such as AWS Lambda.
- In many cases, serverless applications will be cheaper to run and faster to implement. There's also a need to reduce complexity and costs associated with running infrastructure and carrying out development of traditional software systems.
- The reduction in cost and time spent on infrastructure maintenance and the benefits of scalability are good reasons for organizations and developers to consider serverless architectures.

2

First steps to serverless

This chapter covers

- Creation of S3 buckets and Lambda functions
- Writing and deploying Lambda functions
- Using AWS services such as Simple Storage Service, and AWS Elemental MediaConvert
- Using Serverless Framework to organize and deploy functions

To give you an understanding of serverless architectures, you're going to build a small, event-driven, serverless application. Specifically, you are going to build a video transcoding pipeline. You will be able to upload a video file to an S3 bucket and have it transcoded to different resolutions and bitrates. To build this video transcoding pipeline you are going to use AWS Lambda, S3, and AWS Elemental MediaConvert. Later, if you so desire, you can build a website around it, akin to YouTube, but we'll leave it to you as an exercise. If you want to see how we've done it ourselves, you can refer to our first edition that covers the website in some detail.

2.1 Video Encoding Pipeline

At a high level, in this chapter, you are going to learn the following:

- How to construct a rudimentary serverless architecture using three AWS services including Lambda.
- How to use the Serverless Framework to organize and deploy a serverless application.
- How to run, debug, and test the Serverless pipeline you have built in AWS.

Serverless Frameworks

You might have heard that there are several different frameworks that you can use to organize and deploy serverless applications. These include Serverless Application Model (https://github.com/awslabs/serverless-

application-model), Serverless Framework (https://serverless.com), Chalice (https://github.com/aws/chalice), and a few others.

Initially in this chapter you are going to create and deploy functions manually. You will see how to configure a function, deploy, and then run it. Later on, you are going to start using Serverless Framework and learn out how to organize and automate the deployment of our serverless application. We'll continue to use the Serverless Framework throughout the book. You'll see how to use it to create and configure an API and connect various services. Our advice is to always use a framework, like Serverless Framework or Serverless Application Model (SAM). Once you understand the principles of serverless architectures, a framework will accelerate everything you do by leaps and bounds.

Let's talk about the event-driven pipeline you are going to build (let's call it *24 Hour Video)*. Your pipeline will take uploaded videos and encode them to different formats and bitrates. A quick note on AWS costs: most AWS services have a free tier. By following this example, you should stay within the free tier of most AWS services. AWS Elemental MediaConvert, however, is one service that may end up costing you a little bit of money. You are going to use MediaConvert to transcode video files from one format to other formats, resolutions, and bitrates. This service is *pay as you go* without any upfront cost. Pricing is based solely on the duration of the new videos MediaConvert creates and is charged in 10 second increments.

MediaConvert offers two pricing tiers: basic and professional. You are going to use the basic tier in this book, although we invite you to investigate the professional tier if you are going to take your application to the next level (remember us if you end up building the next YouTube!). The basic tier supports features such as single-pass encoding, clipping, stitching and overlays. The professional tier supports quite a few more features. The per-minute rate in the basic tier depends on the resolution and the frame rate of the desired output. It ranges from $0.0075 per minute for basic SD quality output to $0.0450 per minute for UHD output. This rate also differs based on the region you are in. US East 1 (North Virginia), for example, is cheaper than US West 1 (Northern California). This is good, because US East 1 is the region you'll use throughout this book. You can see the tiers and the pricing information at https://aws.amazon.com/mediaconvert/pricing/. Just remember that there's no free tier for MediaConvert, so you'll start paying something almost immediately.

Elastic Transcoder

In the first edition of this book we used a service called Elastic Transcoder (ET) to transcode video (AWS Elemental MediaConvert didn't exist back then). Elastic Transcoder is still a good product that is easy to configure and use. It is also likely to be supported well into the future. If you decide to use ET instead of MediaConvert, you will have to figure out some of the code yourself or refer to the first edition of the book. In terms of pricing, ET offers a free tier with 20 minutes of SD output and 10 minutes of HD (720p or above) output per month (similarly to MediaConvert, a minute refers to the length of the source video, not transcoder execution time). Costs are dependent on the region where Elastic Transcoder is used. In the eastern part of the United States (US East 1) the price for 1 minute of HD output per month is $.03. This makes 10-minute source file cost 30 cents to encode. As a comparison, MediaConvert is priced at $0.0188 per minute for HD output that's between 30 and 60 frames per second. A 10-minute video will end up costing 18.8 cents to encode with MediaConvert in the same eastern region, which is cheaper but, as we said, there's no free tier with MediaConvert. Elastic Transcoder pricing for other regions can be found at https://aws.amazon.com/elastictranscoder/pricing/.

The S3 free tier allows users to store 5 GB of data with standard storage, issue 20,000 GET requests and 2,000 PUT requests, and transfer 15 GB of data out each month. Lambda provides a free tier with 1M free requests and 400,000 GB-seconds of compute time. You should be well within the free tiers of those services with your basic system.

The following are the high-level requirements for 24-Hour Video:

- The transcoding process will convert uploaded source videos to three different resolutions and bitrates:
    - 6Mbps with a 16x9 aspect ratio and a resolution of 1920x1080p.
    - 4.5Mbps with a 16x9 aspect ratio and a resolution of 1280x720p.
    - 1.5Mbps with a 4x3 aspect ratio and a resolution of 640x480p.
- There will be two S3 buckets. Source files will go into the *upload* bucket. Files created by AWS MediaConvert will be saved to the *transcoded* (i.e. encoded from one video format to another) video S3 bucket.

To make things simpler to manage, you'll set up a build and deployment system using the Node Package Manager (npm). You'll want to do it as early as possible to have an automated process for testing, packaging Lambda functions, and deploying them to AWS. But you'll temporarily set aside other developmental and operational aspects such as versioning or deployment and come back to them later.

2.1.1 Amazon Web Services

To create your serverless back end, you'll use several services provided by AWS. These include S3 for storage of files, MediaConvert for video conversion, and Lambda for running custom code and orchestrating key parts of the system. For the most part, you'll use the following AWS services:

- Lambda will handle parts of the system that require coordination or can't be done directly by other services. In this chapter you'll create the first Lambda function that will kick-off MediaConvert jobs. This function will automatically run whenever a file is uploaded to an S3 bucket.
- MediaConvert will encode videos to different resolutions and bitrates. Default presets will remove the need to create custom encoding profiles.

Figure 2.1 shows a detailed flow of the proposed approach. Note that the only point where a user needs to interact with the system is at the initial upload stage. This figure and the architecture may look complex, but we'll break the system into manageable chunks and tackle them one by one over the course of two chapters.

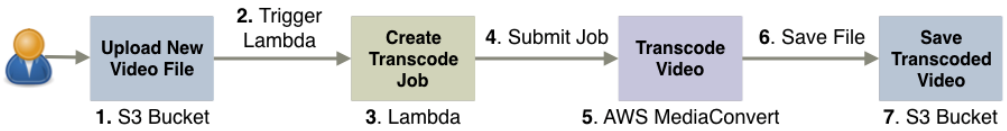


Figure 2.1 This back end is built with S3, MediaConvert, and Lambda. This pipeline may seem to have a lot of steps initially, but we'll break it down, and you'll build a scalable serverless system in no time at all.

2.2 Preparing your system for Serverless

Now is the time to set up services in AWS and install software on your computer. Here's what you'll install on your machine:

- Node.js and its package manager (npm) to help manage Lambda functions and keep track of dependencies.
- AWS Command Line Interface (CLI) to help with deployments, and future use cases & examples.
- Serverless Framework (npm package) to help you organize and deploy your application to AWS.

In AWS you'll create the following:

- An Identity and Access Management user and roles
- S3 buckets to store video files
- The first Lambda function

This section may seem lengthy but it explains a number of things that will help you throughout the book. If you've already used AWS, you'll be able to move through it quickly.

2.2.1 Setting up your system

To begin you need to create an AWS account and install a number of software packages and tools on to your computer. Let's take these in order:

1. Create an AWS account. It's free but you will need to provide your credit card details in case there are any charges (i.e. you go over the free tier). You can create your account at https://aws.amazon.com. We highly recommend that you set up 2 Factor Authentication (2FA) on your account as soon as possible. The instructions are here: https://amzn.to/2ZASm33.
2. After your account is created, download and install the appropriate version of the AWS CLI for your system from http://docs.aws.amazon.com/cli/latest/userguide/installing.html. There are different ways to install the CLI, including an MSI installer if you're on Windows, Pip (a Python-based tool), or using a bundled installer if you're on Mac or Linux.

3. Finally, you need to install Node.js and the Node Package Manager (npm). You can download Node.js from https://nodejs.org/en/download/. The Node Package Manager comes bundled with Node.js.
4. Just a heads up that in a short while you'll need to install Serverless Framework. However, you don't need to do it now. We'll cover it when it is time.

2.2.2 Working with Identity and Access Management

Having an AWS account is fantastic but you cannot do too much with it just yet. For example, the AWS CLI you have just installed isn't going to work. You will not be able to create resources, deploy, or do anything really. To make it work you'll need to create an Identity and Access Management (IAM) user, assign permissions to the user, and then configure the CLI to use the IAM user's credentials. Let's try to do that now:

- In the AWS console, click IAM (Identity and Access Management), click Users, and then click Add user.
- Give your IAM user a name such as `lambda-upload` and select the Programmatic access check box (figure 2.2). Selecting this check box will allow you to generate the access key ID and the secret access key (you'll need these keys for `aws configure` in a few steps).
- Click Next: Permissions to proceed.

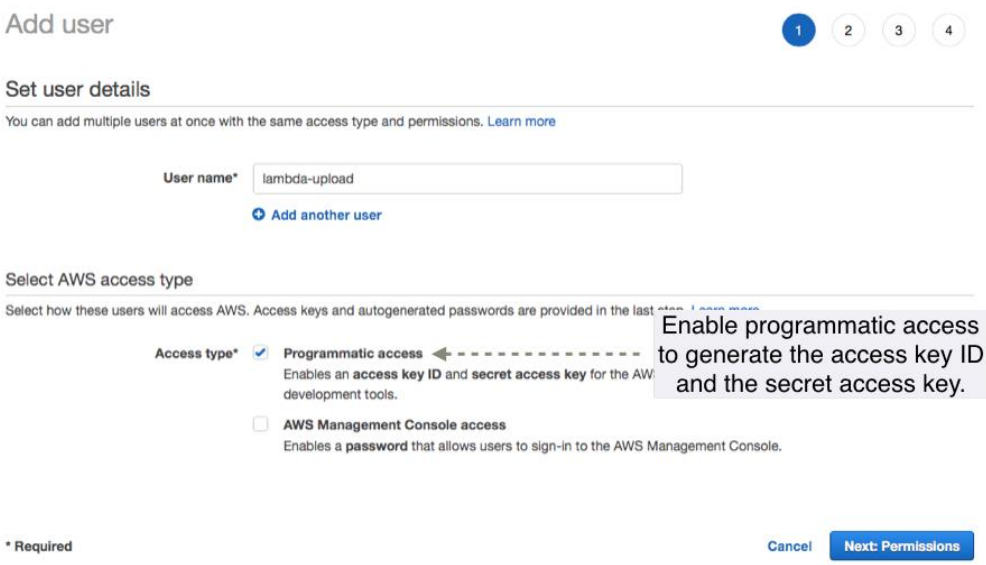


Figure 2.2 Creating a new IAM user is straightforward using the IAM console.

- Select Attach existing policies directly and then click the checkbox next to AdministratorAccess (figure 2.3). Choose Next:Review to proceed.

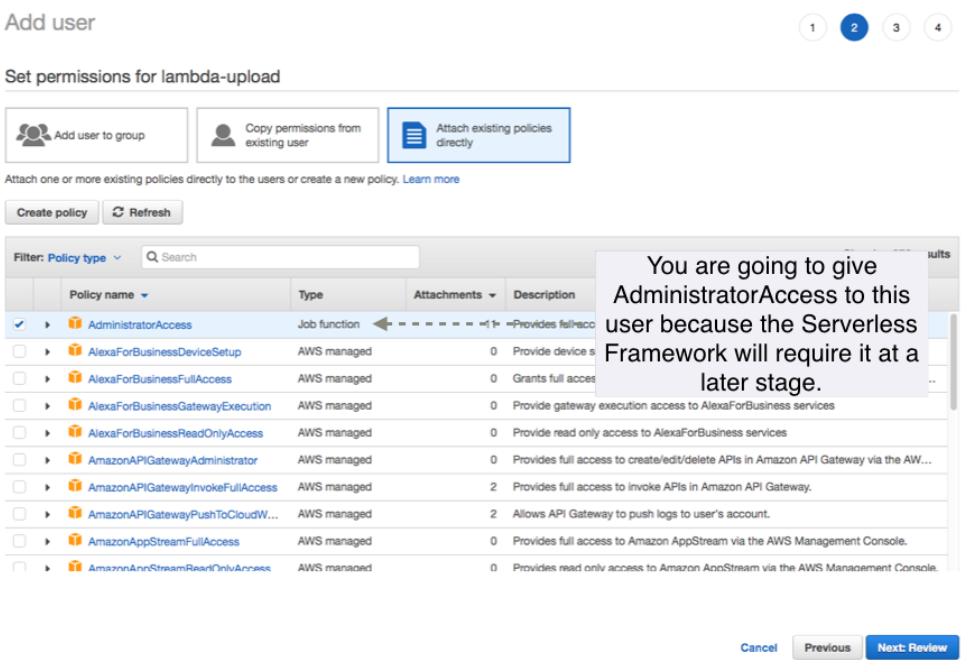


Figure 2.3 Make sure to select the AdministratorAccess policy. You will need it to upload functions and deploy other services.

- On the final page you will be able to review your user details and the permissions summary. Choose Create user to proceed.
- You should now see a table with the username, the access key ID, and the secret access key. You can also download a CSV file with these keys. Download it now to retain a copy of the keys on your computer and click Close to exit (figure 2.4).

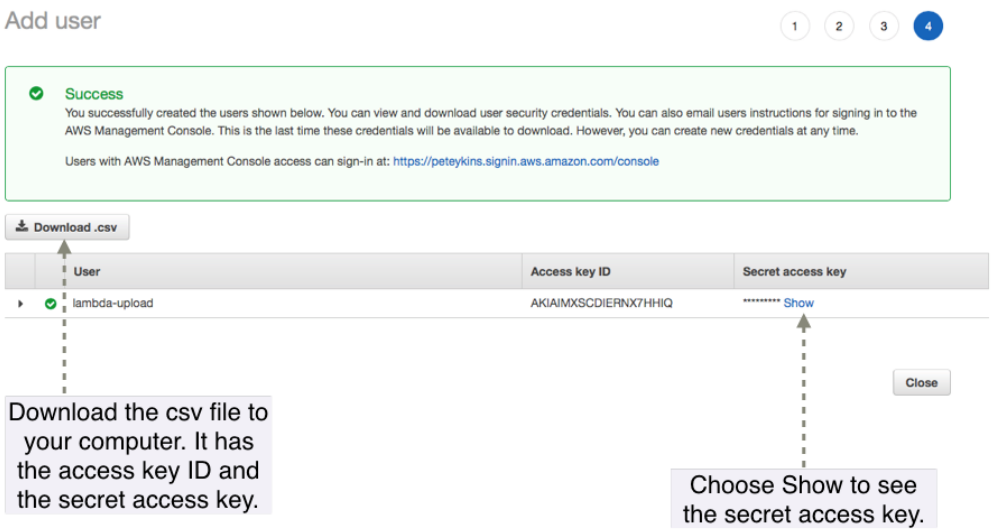


Figure 2.4 Remember to save the access key ID and the secret access key. You won't be able to get the secret access key again once you close this window.

- Run `aws configure` from a terminal on your system. The AWS CLI will prompt for user credentials. Enter the access and secret keys generated for `lambda-upload` in the previous step.
- You'll also be prompted to enter a region, type `us-east-1` and press enter. We recommend that you use the same region for all services (you'll find that it's cheaper and makes things easier to configure). N. Virginia (us-east-1) supports everything we are going to use so for the duration of this book, so make sure to use us-east-1 at all times.
- There will be one more prompt asking you to select the default output format. Set it as `json`.

You are now done with AWS CLI configuration. You created an IAM user and used that user's credentials to configure the CLI on your system. Good job!

Granular Permissions

The best practice when it comes to permissions in AWS is to make them granular. This means that your IAM users and roles should have specific permissions needed to carry out their purpose. They shouldn't have all, administrator-level permissions unless there is a good reason for it. You have just created a lambda-upload IAM user that has

administrator-level permissions. This flies in the face of the advice we've just given. The reason for it is that the Serverless Framework, which you will use shortly, is going to need administrator level access. The Framework calls out to a lot of APIs and it's difficult to configure an IAM user with just the right permissions. If you weren't going to use the Serverless Framework and were going to deploy functions using the AWS CLI then we'd recommend creating an IAM user and assigning a few specific permissions needed to upload your functions.

2.2.3 Let's make a bucket

The next step is to create a bucket in S3. This bucket will contain transcoded videos put there by AWS Elemental MediaConvert. All users of S3 share the same bucket namespace, which means that you have to come up with bucket names that are not in use. In this book and all the examples that follow, we'll assume that this bucket is named something like `serverless-video-transcoded`.

Bucket names

Bucket names must be unique throughout the S3 global resource space. We've already taken `serverless-video-transcoded`, so you'll need to come up with different names. We suggest adding your initials (or a random string of characters) to these bucket names to help identify them throughout the book (for example, `serverless-video-upload-ps` and `serverless-video-transcoded-ps`).

To create a bucket:

- In the AWS console choose S3 and then click Create Bucket.
- Type in a name for the bucket and choose US East (N. Virginia) as the region (figure 2.8).
- Choose Next and continue clicking Next through the wizard until the popup closes (you don't need to specify any additional options at the moment). Your bucket should immediately appear in the console.

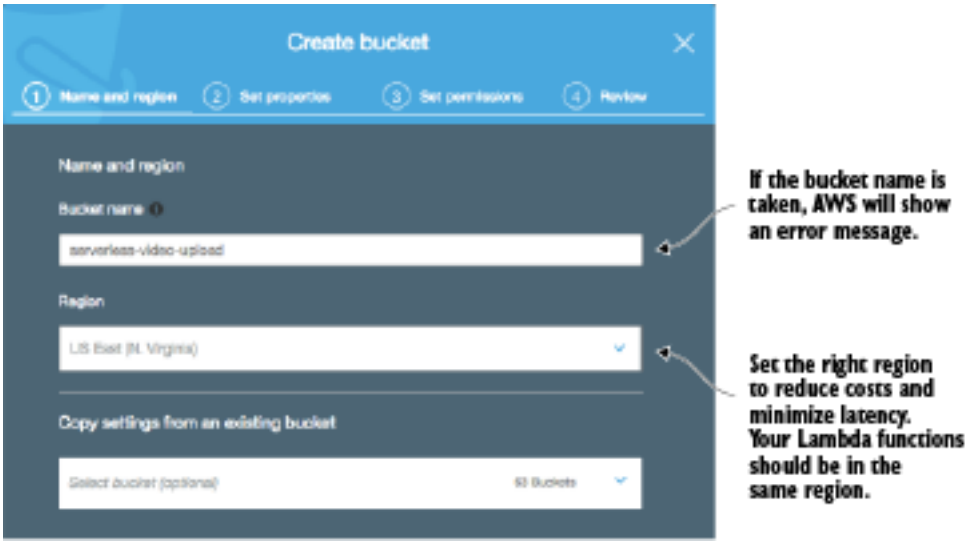


Figure 2.5 Create a bucket from the S3 console. Remember that bucket names are globally unique, so you'll have to come up with your own, new name.

> **NOTE** You are going to end up needing another S3 bucket to which you will upload videos in the first place. The Serverless Framework will create this bucket for you automatically so you don't need to do anything! You could create the transcoded video bucket using Cloud Formation but it is outside the scope of this chapter – it is a good exercise though.

2.2.4 Creating an IAM role

Now you need to create an IAM role for your first Lambda function (you will create this function in a little while). The role will allow your function to interact with S3 and AWS Elemental MediaConvert. You'll add two policies to this role:

- `AWSLambdaExecute` and
- `AWSElementalMediaConvertFullAccess`.

The `AWSLambdaExecute` policy allows Lambda to interact with S3 and CloudWatch. CloudWatch is an AWS service used for collecting log files, tracking metrics, and setting alarms. The `AWSElementalMediaConvertFullAccess` policy allows Lambda to submit new transcoding job to MediaConvert.

- In the AWS console, click IAM, and then click Roles.
- Click the Create role button to begin.
- You will see a list of different AWS technologies under AWS service. Select Lambda and Next: Permissions button.
- In this view, you can search for and attach pre-made policies. Find and attach (by clicking the checkbox on the left) the following two policies:

- AWSLambdaExecute
- AWSElementalMediaConvertFullAccess

- Click Next: Review to proceed to the next page.
- Name your role `transcode-video` and click Create role.

After the role is created, you'll see the list of your existing roles again. Choose `transcode-video` to see what's inside. It should look like figure 2.6.

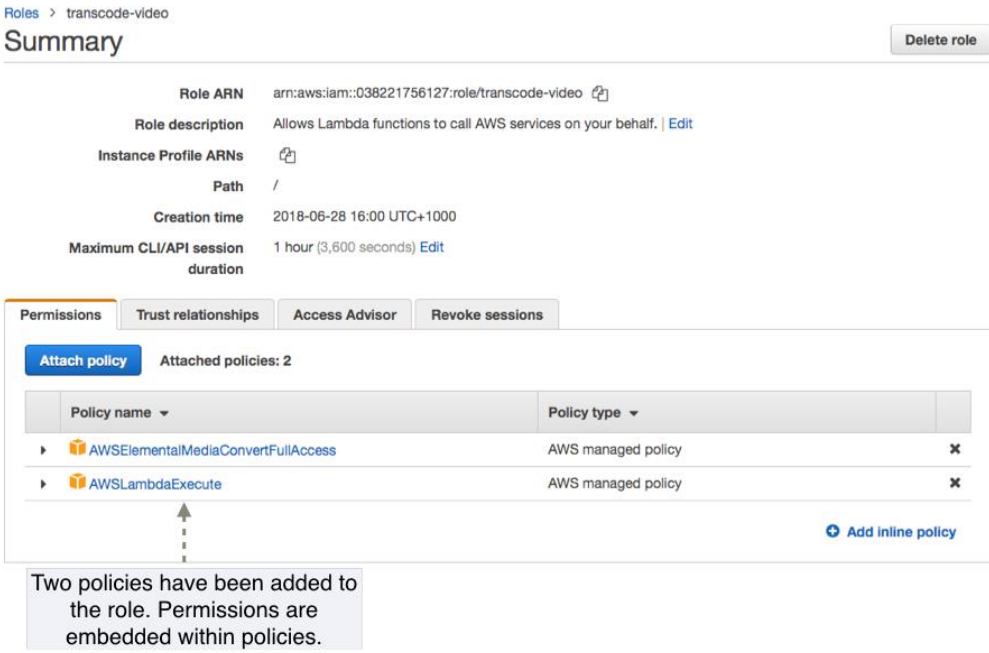


Figure 2.6 Two managed policies are needed for the transcode-video role to access S3 and create AWS Elemental MediaConvert jobs.

2.2.5 AWS Elemental MediaConvert

You are going to use AWS Elemental MediaConvert to convert uploaded video files from one format to another. At a high level, MediaConvert works by taking a file uploaded to an S3 bucket, transcoding the file to one or more different versions, and then placing these versions in to another S3 bucket. When you create a MediaConvert job you'll have to specify this information – including input and output buckets, and what conversion you'll want to carry out. You will also have to specify a MediaConvert endpoint. Each user has a custom endpoint and you need to know where it is. Let's find it now so that you'll know where to look.

- In the AWS console select MediaConvert (it will be under the Media Services category).
- Click the "hamburger" button in the top left corner (it looks like three parallel lines).

- Choose Account from the menu.
- You should see the API endpoint that you will need to use in this chapter (figure 2.7).

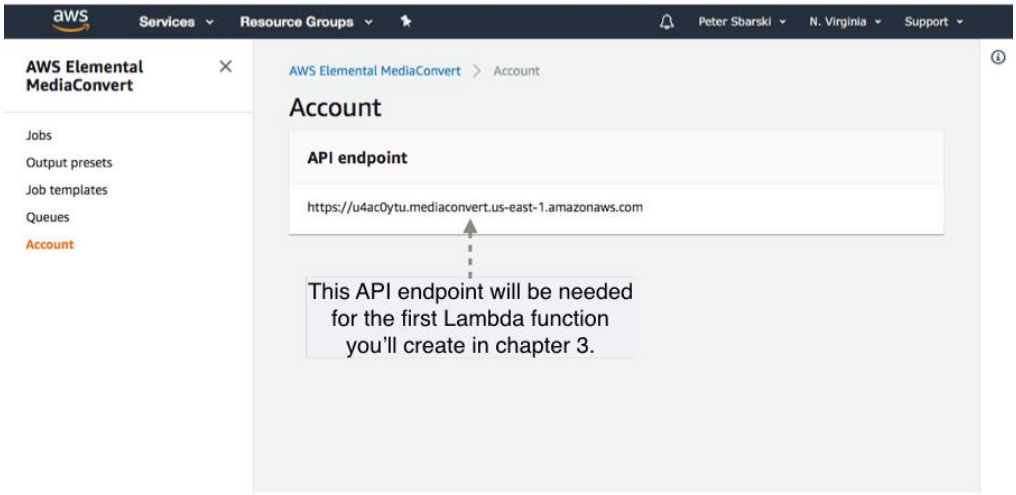


Figure 2.7 You can always refer to these instructions to find the MediaConvert API endpoint if you forget where it is.

Elastic Transcoder

In the first edition of this book we used the Elastic Transcoder service instead of MediaConvert. If you still want to use Elastic Transcoder you will have to create and configure a pipeline. Here's how you would do that (just remember that you don't need to do this if you are going to use MediaConvert):

1. In the AWS console click Elastic Transcoder and then click Create a New Pipeline.
2. Give your pipeline a name, such as 24 Hour Video, and specify the input bucket, which in our case is the upload bucket (`serverless-video-upload`).
3. Leave the IAM role as it is. Elastic Transcoder creates a default IAM role automatically.
4. Under Configuration for Amazon S3 Bucket for Transcoded Files and Playlists, specify the transcoded videos bucket, which in our case is `serverless-video-transcoded`. The Storage Class can be set to Standard.
5. You're not generating thumbnails but you should still select a bucket and a storage class. Use the second, transcoded videos bucket for it again.
6. Click Create Pipeline to save.

One difference between MediaConvert and Elastic Transcoder is that with Elastic Transcoder you must create a pipeline and specify an input and output buckets. MediaConvert makes this a lot easier because you can specify all buckets programmatically.

You are nearly there but there's one more IAM role you need to create right now to make things easier later.

2.2.6 MediaConvert Role

You need to create a role for the MediaConvert service. MediaConvert needs to have access to S3 as well as the API Gateway. Without this role, MediaConvert will simply not run when you try to invoke it from Lambda. To create the role, follow these steps (or try it on your own!):

- In the AWS console, click IAM, and then click Roles.
- Click the Create role button to begin.
- You will see a list of different AWS technologies under AWS service. Select MediaConvert and Next: Permissions button.
- AWS has already predefined what policies you need for this role. These S3 Full Access and API Gateway Full Invoke. Choose Next: Tags.
- Click Next: Review to proceed to the next page.
- Name your role `media-convert-role` and click Create role.

Copy the ARN of the role to a notepad or somewhere you can easily retrieve. You'll need the role ARN as well as the API endpoint in listing 2.3.

2.3 Starting with the Serverless framework

The Serverless Framework is going to help you organize your functions and deploy them to AWS. This framework is powerful so you will get a lot of flexibility in terms of how to package code, what variables to use, and environments to deploy to. Later on, you'll set up a CI/CD pipeline that will deploy the entire system every time you push a change to GitHub. If you get stuck with the Serverless Framework go to appendix C or https://serverless.com/framework/docs. Appendix C features a thorough walkthrough the Framework as well as useful hints and tips. However, if you don't find your answer there then the online documentation is the way to go.

2.3.1 Setting up Serverless framework

Install the Serverless Framework by running `npm install -g serverless` from the terminal.

CREDENTIALS

The Serverless Framework needs access to your AWS account so that it can create and manage resources on your behalf. To let the Serverless Framework access your AWS account, you're going to need the access key ID and the secret access key you created in section 3.2.2.

HELLO WORLD!

Having installed the Serverless Framework and configured credentials let's test that it works. In your terminal run the following command:

```
serverless create --template aws-nodejs --path hello-world
```

Change to the newly created directory by running `cd hello-world`. You should see two files in this directly serverless.yml and handler.js. The first file is a project file (a service) that describes functions, events and resources that the function may use. The second file is an example Lambda function that you can change! Open handler.js and modify the implementation of the function to be the same as listing 2.1.

Listing 2.1 A new hello-world Lambda function

```
'use strict';

module.exports.hello = async event => {            #A
  const response = {
    statusCode: 200,
    body: JSON.stringify({
      message: 'Go Serverless v1.0!
  [CA]Your function executed successfully!',
      input: event,
    }),
  };

  return response;
};
```

#A This is a very basic function. You could run it now in AWS and see a message.

Once you have finished modifying the function, remember to save, and close. You are now ready to deploy so run `serverless deploy` from the terminal and hit enter (make sure you are in the same directory as the serverless.yml file before you deploy, otherwise you'll get an error message).

You'll see the Serverless Framework package up files, prepare a CloudFormation Stack, and deploy your function to AWS. As soon as the deployment as finished, you'll see a bit of useful information such as the stage used for the function (dev), the region (us-east-1), and the name of the service (hello-world). Your function will be called `hello-world-dev-hello` a combination of the service name, stage, and the function export. If you don't like the way your deployed Lambda function is named, you can set a different name in serverless.yml using the name parameter.

You can finally check that the function was successfully deployed by opening the Lambda console in AWS and running the function from there. Another option is to invoke the deployed function from the terminal. The function will run in AWS and return a response. Try that by running the following command in the terminal: `serverless invoke --function hello`. The Serverless Framework will know to invoke this function from the cloud environment.

Serverless Deploy

You don't need to type out serverless (as in serverless deploy) each time you want to deploy or do an operation. You can use the `sls` abbreviation. So, the following commands are entirely valid: `sls deploy` or `sls invoke --function hello`.

2.3.2 Bringing Serverless Framework to 24-Hour Video

Now that you have the Serverless Framework to work let's get busy with 24-Hour Video. You are going to create a new function and reference it in serverless.yml. This will allow you to deploy this function (and then additional functions) with a single command and sustainably grow and organize your entire serverless application.

1. In a terminal window run the following command `sls create --template aws-nodejs --path twentyfour-hour-video` (The reason we used "twentyfour" instead of "24" is because a service name must begin with an alphabetic character).
2. Change to the new twentyfour-hour-video directory that was just created.
3. Delete handler.js but leave serverless.yml intact.
4. Create a new subfolder called `transcode-video`.

In a moment you'll begin changing serverless.yml. You can stick to our implementation (listing 2.2), however, there are 3 parameters that you must change to correctly reflect your environment. These parameters are bolded in listing 2.2 and are:

- The name of the upload bucket
- The name of the transcoded video bucket
- Role ARN for your function

TRANSCODED VIDEOS BUCKET

In listing 2.2 there's a custom property called *transcode-video*. This property contains the name of your *transcoded* video bucket – this is the bucket that you have manually created in S3. Update this property to be the name of the bucket you created in section 3.2.2.

UPLOAD BUCKET

In listing 2.2 you must specify the name of the *upload* bucket. This is a new bucket that doesn't yet exist, but Serverless Framework will create it for you. Remember, you need to come up with a bucket name that is globally unique. One way to do it is to prefix or post-fix your full name (unless you have a common name) or a few random letters and numbers. Serverless Framework, through CloudFormation, will create the bucket for you automatically. You can try to go for something like *upload-bucket-firstname-lastname*. If the bucket name is already taken, during deployment Serverless Framework will tell you; you'll be able to change it and try again.

ROLE ARNS

You must specify an IAM role for the function. Luckily, you have created a role in section 3.2.4. You need to find the ARN of that role, copy it and update the parameter called *transcode-video-role*. To get the role ARN and update serverless.yml follow these easy steps:

1. In the IAM console select Roles
2. Find the *transcode-video* role and select it
3. Copy the value for the Role ARN

4. Paste the value in to serverless.yml for the transcode-video-role

Replace the contents of serverless.yml with listing 2.2 and don't forget to update the upload, transcode buckets, as well as the role.

Listing 2.2 A new hello-world Lambda function

```
service: twentyfour-hour-video

provider:
  name: aws                                                  #A
  runtime: nodejs12.x
  region: us-east-1                                  #B

package:
  individually: true                                         #C
  excludeDevDependencies: true
  exclude:
    - '*/**'
    - '**'

custom:
  upload-bucket: upload-video-bucket                                  #D
  transcode-bucket: transcoded-video-bucket                  #E
  transcode-video-role: arn:aws:iam::038221756127:role/transcode-video  #F

functions:
  transcode-video:
    handler: transcode-video/index.handler
       role: ${self:custom.transcode-video-role}
    package:
      include:
        - transcode-video/**
    environment:
       TRANSCODED_VIDEO_BUCKET: ${self:custom.transcode-bucket}
    events:
        - s3: ${self:custom.upload-bucket}                   #G
```

#A The provider is AWS; however, Serverless Framework supports other providers like Azure and Google Cloud. So, you could create and deploy functions to those cloud environments too.

#B Here we are explicitly defining the region you'll be deploying to. You can override this setting and deploy to other regions.

#C This tells the Serverless Framework to package each function individually. This is a good thing to do as we don't want to co-mingle all functions together. If you don't do this then everything will still work but your deployment packages are going to be larger and will have all functions included in them.

#D This is a custom variable that allows us to replace config values. This particular variable should be set to the name of your upload bucket that you are creating for the first time here. Note that the name of the bucket must be globally unique.

#E The transcode-bucket should be set to the name of your transcoded videos bucket. You have created this bucket manually in section 3.2.2.

#F This is the transcode-video role ARN you created earlier in section 3.2.4. Update the ARN to your value.

#G Here you are specifying the event trigger for this Lambda function. It will be the S3 upload bucket.

2.3.3 Creating your first Lambda function

Now that you've created a serverless.yml file, in your terminal window, change to the transcode-video folder, and run `npm init`. Agree to all the options by hitting enter. You can change anything you want. It will not affect your function. The end result of running `npm init` is that you'll get a file called package.json. This file is needed to record information about additional dependencies and libraries your function may use later.

Here's what is going to happen to your first Lambda function:

- The function will be invoked from S3 as soon as a new file is placed in a bucket.
- Information about the uploaded video will be passed to the Lambda function via the event object. It will include the bucket name and the key of the file being uploaded.
- This function will prepare a transcoding job for AWS MediaConvert. It will specify the input file and three new outputs.
- Finally, the function will submit the job to MediaConvert and write a message to an Amazon CloudWatch Log stream.

Create a new file named index.js and open it in your favorite text editor. This file will contain the first function. The important thing to know is that you must define a function handler, which will be invoked by the Lambda runtime. The handler takes two parameters—`event` and `context`—and is defined as follows:

```
exports.handler = async function(event, context){}
```

Listing 2.3 shows this function's implementation. Copy this listing into index.js. Before you can deploy and run this code though, you'll need to make a few small changes as detailed in the text after the code listing.

Listing 2.3 Transcode video Lambda

```
'use strict';

const AWS = require('aws-sdk');
const mediaConvert = new AWS.MediaConvert({
    endpoint: 'https://u4ac0ytu.mediaconvert.us-east-1.amazonaws.com'  #A
});

const outputBucketName = process.env.TRANSCODED_VIDEO_BUCKET;          #B

exports.handler = async function(event, context) {
    const key = event.Records[0].s3.object.key;
    const sourceKey = decodeURIComponent(key.replace(/\+/g, ' '));
    const outputKey = sourceKey.split('.')[0];

    console.log(key, sourceKey, outputKey);

    const input = 's3://' + event.Records[0].s3.bucket.name + '/' +
       [CA]event.Records[0].s3.object.key;
    const output = 's3://' + outputBucketName + '/' + outputKey + '/';

    console.log(input, output);

    try {
```

```
        const job = {
            "Role": "arn:aws:iam::038221756127:role/media-convert-role", #C
            "Settings": {
                "Inputs": [{
                    "FileInput": input,                                  #D
                    "AudioSelectors": {                                  #E
                        "Audio Selector 1": {
                            "SelectorType": "TRACK",
                            "Tracks": [1]
                        }
                    }
                }],
                "OutputGroups": [{
                    "Name": "File Group",
                    "Outputs": [{
                        "Preset": "System-
      [CA]Generic_Hd_Mp4_Avc_Aac_16x9_1920x1080p_24Hz_6Mbps",
                        "Extension": "mp4",
                        "NameModifier": "_16x9_1920x1080p_24Hz_6Mbps"
                    }, {
                        "Preset": "System-
      [CA]Generic_Hd_Mp4_Avc_Aac_16x9_1280x720p_24Hz_4.5Mbps",
                        "Extension": "mp4",
                        "NameModifier": "_16x9_1280x720p_24Hz_4.5Mbps"
                    }, {
                        "Preset": "System-
      [CA]Generic_Sd_Mp4_Avc_Aac_4x3_640x480p_24Hz_1.5Mbps",
                        "Extension": "mp4",
                        "NameModifier": "_4x3_640x480p_24Hz_1.5Mbps"
                    }],
                    "OutputGroupSettings": {
                        "Type": "FILE_GROUP_SETTINGS",
                        "FileGroupSettings": {
                            "Destination": output                        #F
                        }
                    }
                }]
            }
        };

        const mediaConvertResult = await [CA]mediaConvert.createJob(job).promise();

        console.log(mediaConvertResult);

    } catch (error) {
        console.error(error);
    }
};
```

#A You must specify your personal MediaConvert endpoint for the service to work. Where to find the endpoint will be shown right after this.

#B The name of your transcoded videos bucket will be coming from the serverless.yml file. This is how you reference an environment variable using node.js.

#C You must provide a role ARN for MediaConvert to operate with. You created this role in section 3.2.6.

#D The MediaConvert job definition must specify where the input files are held. This is the syntax for doing so.

#E The MediaConvert job definition must also specify the Audio Selector. You'll default to specifying a single audio track in the video.

#F The OutputGroups section is required. It's used to tell MediaConvert what type of output to create. In this example you'll be creating three different output files using the inbuilt MediaConvert presets.

Source code at your fingertips

Our GitHub repository at https://github.com/sbarski/serverless-architectures-aws has all the code snippets and listings you need for this book. So you don't have to manually type anything out—unless you really want to.

MEDIACONVERT ENDPOINT

At the very top of the function in listing 2.3 you'll find a line that declares the MediaConvert endpoint. You must modify this code and set your own endpoint. To get this endpoint refer to section 3.2.5 or follow these steps:

1. In the AWS console select MediaConvert (it will be under the Media Services category).
2. Click the "hamburger" button in the top left corner (it's the button that looks like three parallel lines).
3. Choose Account from the menu. You'll see the API endpoint that you should copy in to the code.

MEDIACONVERT ROLE

In section 3.2.6 you created an IAM role for the MediaConvert service. You need to specify the ARN of this role for MediaConvert to run. In listing 2.3. you'll need to specify the ARN of that role. Make sure to look it up and copy it over correctly. Make sure not to confuse the two IAM roles that you have. The IAM role created in section 3.2.4 is intended for the *transcode-video* Lambda function. The role created in 3.2.6 is intended for MediaConvert.

MEDIACONVERT OUTPUTS

The function in listing 2.3 declares three new outputs that define the format for your newly transcoded videos (this includes bitrate, resolution and so forth). The templates specified in the listing 2.3 are generic templates built in to MediaConvert. Luckily, you aren't forced to use the ones we've selected; you can select different templates or even create your own. To have a look at other available presets in MediaConvert do the following:

1. In the AWS console select MediaConvert.
2. Click the "hamburger" button in the top left corner.
3. Choose Output presets.
4. From the dropdown that says Custom Presets select System Presets.
5. You'll see a grid of different available presets you can use (figure 2.8). Note that the grid has multiple pages and that you can choose to see different categories (MP4, HLS, Broadcast-XDCAM, and so on).

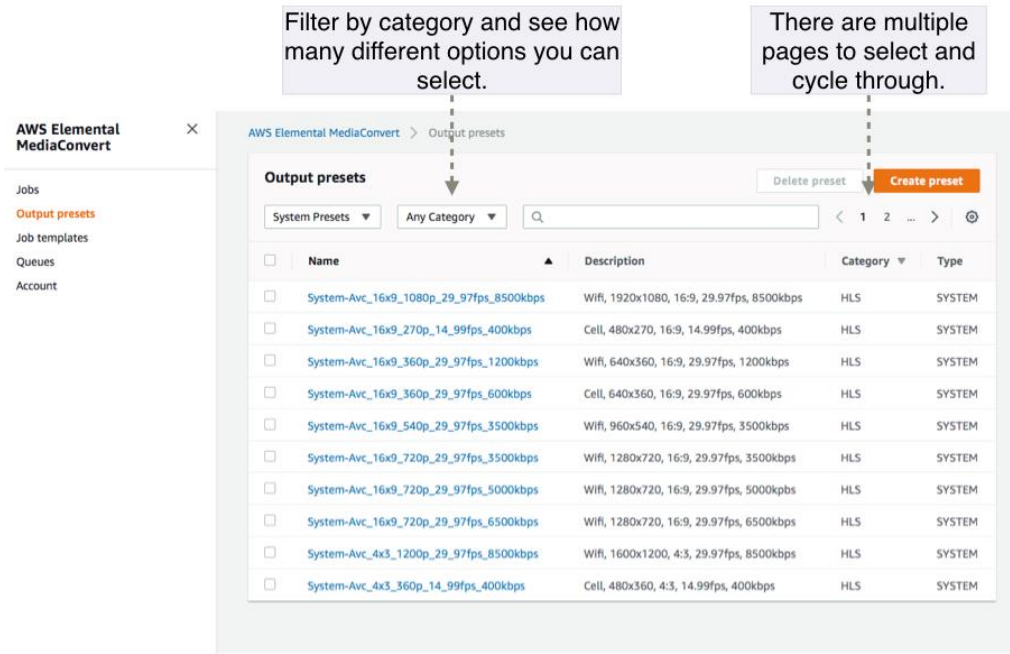


Figure 2.8 The MediaConvert presets page allows you to select system presets or configure your own.

If you want to create a different type of video using the code in listing 2.3, select the name of the desired preset, and copy it in to the function like you've done for the others (remember to specify the extension and the name modified). As an example, if you wanted to add an HLS output, you'd need to add something like this to the Outputs array in the function:

```
{
   "Preset": "System-Ott_Hls_Ts_Avc_Aac_16x9_1280x720p_30Hz_3.5Mbps",
   "Extension": "hls",
   "NameModifier": "_Hls_Ts_Avc_Aac_16x9_1280x720p_30Hz_3.5Mbps "
}
```

DEPLOYMENT

Deploy your first function from the terminal by typing `sls deploy`. Your deployment should succeed and you should see your functions in AWS. The first function is going to be named something like twentyfour-hour-video-dev-transcode-video. Later, if you want, you can remove this function from AWS by running `serverless remove` from the terminal.

One other note: the deployment process may create an additional bucket named something like: twentyfour-hour-video-de-serverlessdeploymentbuck-sq06y6wjku9z. This is normal. This bucket is created by Serverless Framework and is used to upload the CloudFormation templates it generates. You can safely ignore this bucket – do not manually delete. The `serverless remove` command will remove it for you.

2.3.4 Testing in AWS

To test your first function in AWS, upload a video to the upload bucket. Follow these steps:

1. Jump in to the S3 console.
2. Click into the video upload bucket and then select Upload (figure 2.9).
3. You'll see an upload dialog appear on your page. Click Add Files, select a file from your computer, and click the Upload button. All other settings can be left as they are.

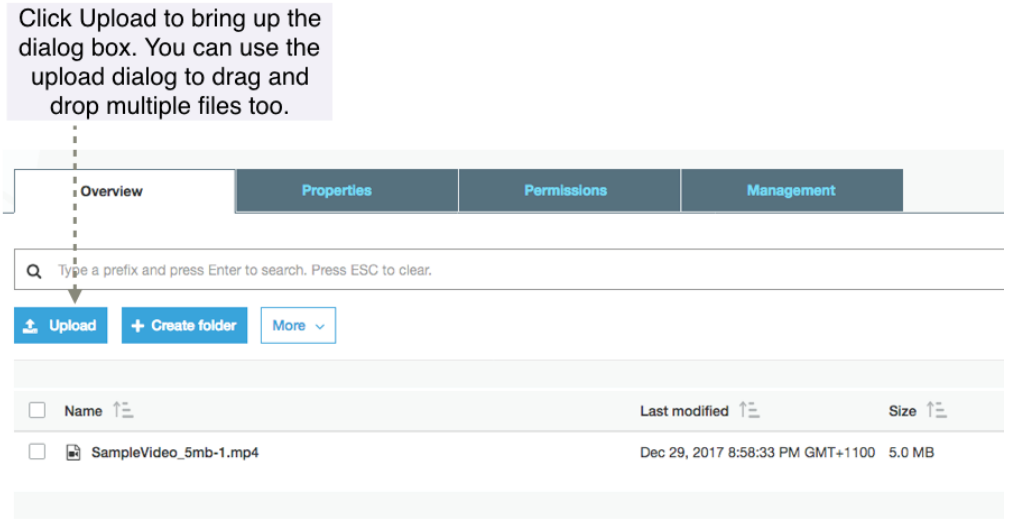


Figure 2.9 It's better to upload a small file initially because it makes the upload and transcoding a lot quicker.

After a time, you should see three new videos in your `transcoded videos` bucket. These files should appear in a folder rather than in the root of the bucket (figure 2.10). The length of time to produce a new video depends on the duration of the file you've uploaded. It may take 5 minutes (or even longer) to produce a new file so grab a cup of tea while you wait.

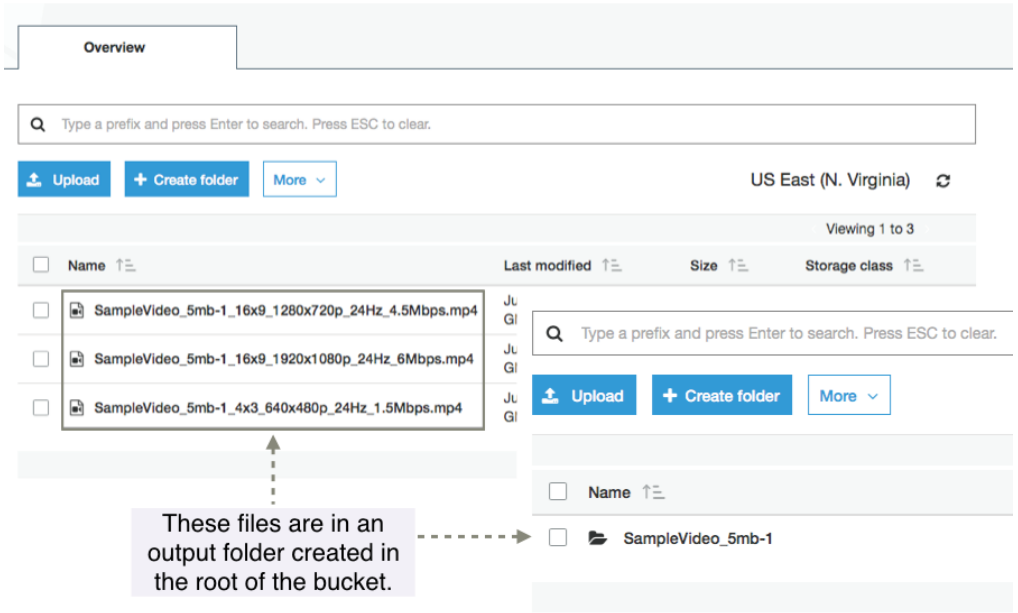


Figure 2.10 MediaConvert will generate three new files and place them in a folder in the transcoded videos S3 bucket.

2.3.5 Looking at logs

Having performed a test in the previous section, you should see three new files in your very own `transcoded video` bucket. But things may not always go as smoothly (although we hope they do)! In case of problems, such as new files not appearing, you can check two different logs for errors. The first and most important is Lambda's log in CloudWatch. To see the log, perform the following steps:

1. Choose Lambda in the AWS console and then click your function name.
2. Choose the Monitoring tab. You should see different graphs with numbers.
3. One of those graphs will be labeled *Error count and success rate*. If there is a spike (that is, the count is more than 0), it means that there is a problem.
4. Click View logs in CloudWatch to open CloudWatch.
5. You'll see all log entries ordered by date. On the right you'll see which stream they belong to. You can click on each log entry to see more detail including error messages.
6. If you previously saw that your Invocation error rate was more than zero, try to find the log entry with the error and fix the problem.

If Lambda logs reveal nothing out of the ordinary, take a look at the AWS MediaConvert logs:

1. Click MediaConvert in the AWS console.
2. Choose the "hamburger" button in the left to open the sidebar.
3. Choose Jobs from the menu.
4. On the right you should see a list of job. You can click into a job to see more information if it failed (figure 2.11).

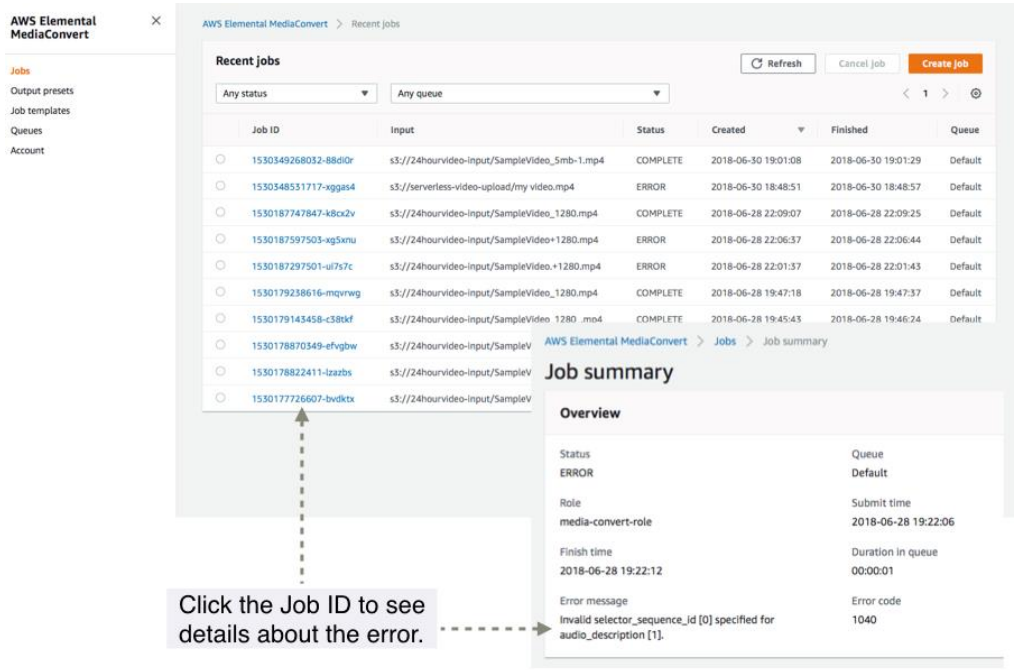


Figure 2.11 MediaConvert failures can occur for a variety of reasons, including the source file being deleted before the job started, an error with the code in the Lambda function, or misconfiguration.

When problems happen

In our experience problems often occur because IAM permissions haven't been configured correctly or there was a typo somewhere in your function code. AWS doesn't always have the most descriptive error messages so, sometimes, a bit of digging around and investigative work with CloudWatch is required.

2.4 Lessons learned

We learnt the following lessons in this chapter:

- The best way to organize Serverless applications is to use Infrastructure as Code

frameworks like the Serverless Framework or the Serverless Application Model (SAM). Deploying functions and setting up services manually is great for learning but is not sustainable in the long-term. Serverless Framework and SAM can help to organize and deploy even the most complex Serverless applications.

- Serverless applications and pipelines usually consist of different services connected together. In the 24-Hour Video example you used Lambda, S3 and AWS Elementals MediaConvert. Most Serverless applications will use a combination of services.
- CloudWatch is an important AWS services for logging what happens within your AWS Lambda functions. It's vital for you to learn how to use it as you are most definitely going to use it.
- Security in AWS is controlled primarily via IAM (although there are some exceptions). If you wish to become an expert at AWS and Serverless, knowing how IAM works is essential.
- Estimating cost in AWS can be very tricky. A lot of services have generous free tiers but can end up costing a lot if used incorrectly. Make sure you review the costs of all services you use and understand what the potential cost can be.

4

Yubl case study – architecture highlights, lessons learned

This chapter covers

- The original Yubl architecture and its problems
- The new architecture on serverless and the architectural decisions behind it
- Strategies and patterns for moving an existing monolith application to serverless
- Lessons learned from this migration

In April 2016, I joined a social network based in London called Yubl. There I inherited a monolithic backend system written in Node.js and running on a handful of EC2 instances. The original system had taken 2.5 years to implement and had a long list of performance and scalability issues once it went live. With a small team of six engineers, we managed to move the platform to serverless over the course of 6 months. Along the way, we added many new features and addressed the existing performance and scalability issues. We reduced feature delivery time from months to days, and in some cases, hours. And though cost was not the main motivation for undertaking this transformation, our spending on the serverless architecture was roughly 5% of what we spent on EC2.

4.1 The original architecture

Yubl (short for "your social bubble") was a mobile-first, social network designed for the 17 to 25-year-old demographic. The user-generated posts (called *yubls*) can contain videos, as well as animated and interactive elements. And the app had all the social features you can find in other social networks – follow users, private and group chat, liking and resharing content, and others (figure 4.1).

Figure 4.1 Screenshots of the Yubl mobile app

The original architecture consisted of the following (figure 4.2):

- A monolithic REST API written in Node.js and running on EC2
- A WebSockets API written in Node.js and running on EC2
- A monolithic MongoDB database hosted in MongoLab
- A CloudAMQP message queue
- A cluster of background workers written in Node.js and running on EC2

What is MongoDB and MongoLab?

MongoDB is a popular document-oriented NoSQL database that allows you to store JSON documents. You can learn more about it at https://www.mongodb.com.

MongoLab is an online service that provides MongoDB hosting as a service. You can create a MongoDB cluster with a few clicks and MongoLab takes care of the underlying infrastructure for you. You can learn more about it at https://mlab.com.

What is RabbitMQ and CloudAMQP?

Advanced Message Queueing Protocol (AMQP) is an application protocol for message-oriented middlewares. It supports message queueing and routing and is often used in publish-and-subscribe systems.

RabbitMQ is an open-source message broker that implements the AMQP protocol.

CloudAMQP is an online service that provides RabbitMQ hosting as a service. You can learn more about it at https://cloudamqp.com.

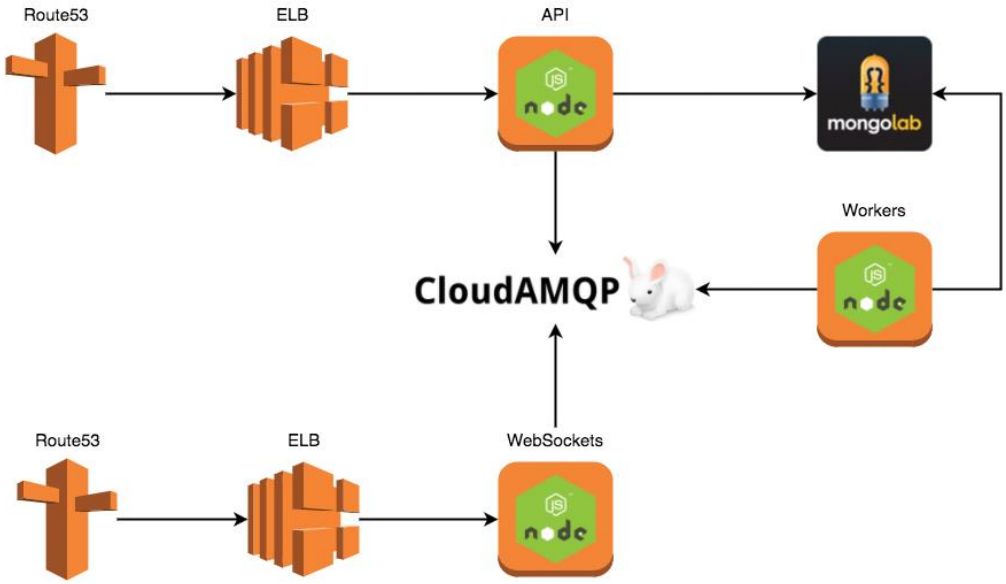


Figure 4.2 A high-level overview of the original architecture.

4.1.1 Scalability problems

Being an early stage social network, the baseline traffic at Yubl was very low, but they managed to attract several high-profile Instagram influencers onto the platform. These influencers brought many of their Instagram followers along. Many of these influencers had tens of thousands of followers and drove unpredictable and spiky traffic through the system whenever they posted new contents.

We often saw 100x spikes in traffic as thousands of users flooded in all at once to see their favorite influencer's new content. These traffic spikes were usually short-lived, which was problematic for our EC2-based system because EC2 auto-scaling can't react fast enough. It typically takes EC2 instances a few minutes to spin up. By the time they are ready to serve user requests, it would have been too late. The traffic spikes have come and gone and many users would have left after having experienced a laggy response time.

As a workaround, we had to run a much larger EC2 cluster and scale up much earlier than we wanted. This resulted in a lot of waste because we had to pay for lots of EC2 resources that we were not using. Our cluster of API webservers had an average utilization of just 2% to 5%.

4.1.2 Performance problems

The monolithic MongoDB database was also a constant source of performance and scalability problems. Every read and write operation hit the database directly; some API operations can take a heavy toll on a MongoDB server. One example of these is user search, which was a frequently used API call and executed a complex regex query against MongoDB.

Other examples included user recommendations, which executed a complex query to find second- and third-degree connections to the current user—that is, those who follow your followers, or those followed by users you follow.

4.1.3 Long feature delivery cycles

The codebase was complex, and many features were intertwined through shared MongoDB tables and implicit coupling through shared libraries.

Although there were plenty of unit tests with a reasonable code coverage, these did not prove useful because code changes often passed all the tests, only to fail when deployed to the AWS environment. The interaction with external services were mocked thoroughly and therefore not covered by the tests. In many cases, the tests simply confirmed the mocks were working and returned what was requested even if the MongoDB query contained syntax errors. We had little faith in the tests because they gave us too many false-positives.

To make matters worse, every deployment required taking the whole system down for 30 minutes or more during which time users received no feedback and the app just appeared broken.

Features used to take months to go to production. Even simple changes often took weeks to complete, which was frustrating to everyone involved.

4.1.4 Why serverless?

Based on the requirements for our system and the problems the current implementation experiences, serverless was a great fit for the following reasons:

- AWS Lambda auto-scales the number of concurrent executions based on load. This happens instantly and handles those unpredictable spikes we experience effortlessly.
- AWS Lambda deploys functions to three Availability Zones by default, which provides significant redundancy without incurring extra costs. We pay only when a function runs, whereas with EC2, we paid for the redundancy in a multi-AZ setup, which also dilutes the traffic and reduces the resource utilization even further.
- AWS manages the underlying physical infrastructure as well as the operating system that our code runs on. They apply patches and security updates regularly and do a much better job of keeping the OS secure than we can. This removes a whole class of vulnerabilities that plague so many software systems around the world.
- With tools such as the Serverless framework, the deployment pipeline for our application is drastically simplified.
    - A typical deployment takes less than a minute and has no downtime because AWS Lambda automatically routes requests to the new code.
    - Most importantly, it allows us to focus on addressing core business needs—almost every line of our code is business logic—and it allows the development team to move quickly, knowing that what they build is scalable by default.
- The number of production deployments went from four to six per month to averaging more than 80 per month with the same sized team. We didn’t have to hire more people to go faster, we allowed each developer to be more productive instead.

- As we migrated more and more of the system to serverless, scalability, cost and reliability all improved. There were far fewer production issues, and we were spending a fraction of what we spent on EC2 previously.

4.2 The new serverless architecture

By November 2016, less than 8 months after I joined the company and started us on the journey to serverless, almost the entire backend system was migrated to serverless, using a combination of services such as API Gateway, Lambda, DynamoDB, Kinesis and so much more. Along the way, we enhanced existing features and implemented countless new features. We also addressed many security issues with the previous system. Over the system's reliability increased drastically. We experienced only one minor outage to our production environment because of a brief S3 outage.

When finished, our system looked something like figure 4.3.

The following points are some key highlights of the new serverless architecture on AWS:

- The monolith was broken up into many microservices.
- Every microservice has its own GitHub repo and one Continuous Integration/Continuous Delivery (CI/CD) pipeline. All its components are deployed together as one CloudFormation stack, using the Serverless framework.
- Most microservices have a REST API running under its own subdomain, such as `search.yubl.com`.
- Every microservice has its own database for the data it needs to function. Most use DynamoDB, but it's not universal because different microservices have different data needs.
- Every state change in the system is captured as an event and published to a Kinesis Data Stream. For example, a user created, a user posted a Yubl, and so on.
- Most of the time, we prefer to synchronize data between microservices through events rather than synchronous API calls at runtime. This helps prevent cascade failures when one microservice experiences an outage in production. Instead, microservices subscribe to the relevant Kinesis Data Stream and copy needed data from the appropriate events.

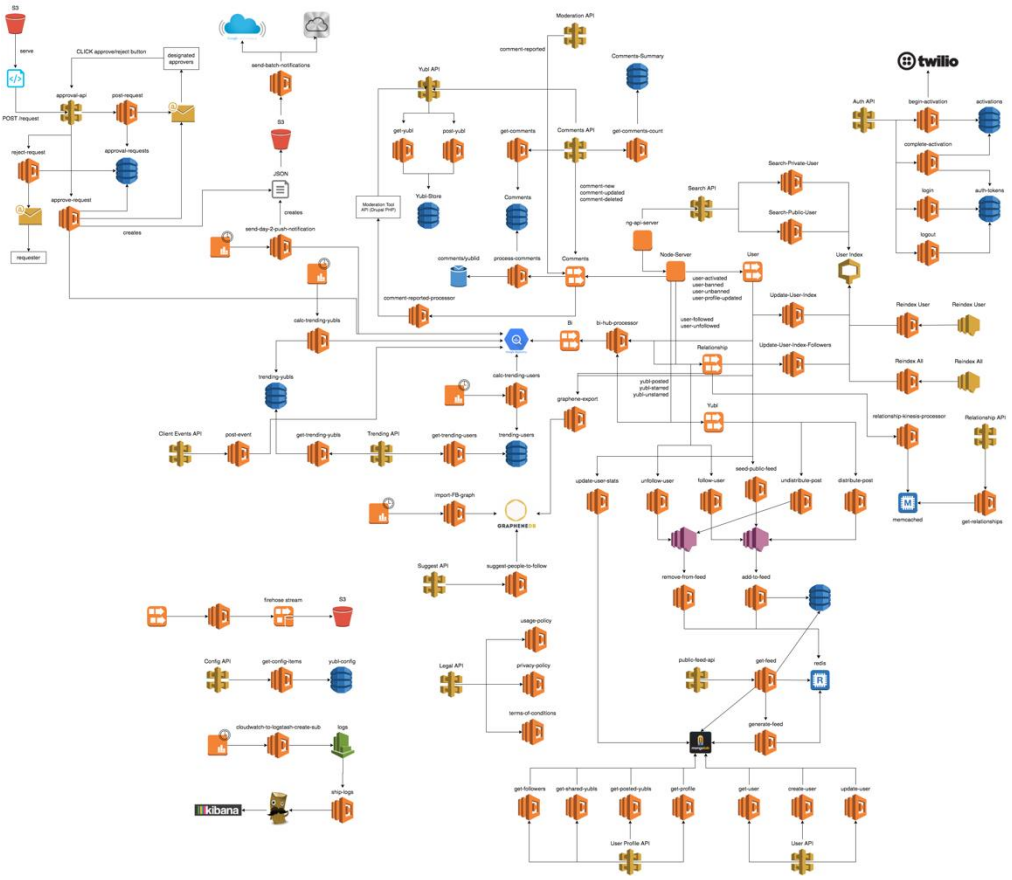


Figure 4.3 A high-level overview of the new architecture running on serverless components.

Now let's dive into one specific example of how we extracted the search feature out of the monolith and built a microservice around it with its own REST API.

4.2.1 The new search API

One of the first and most important steps was to ensure that our legacy monolith would publish its state changes to Kinesis Data Streams. This gave us a foundation to build the new microservices by building on top of these events. To extract the search capability out of the monolith, we created a new search microservice. Figure 4.4 shows the high-level architecture of this search microservice.

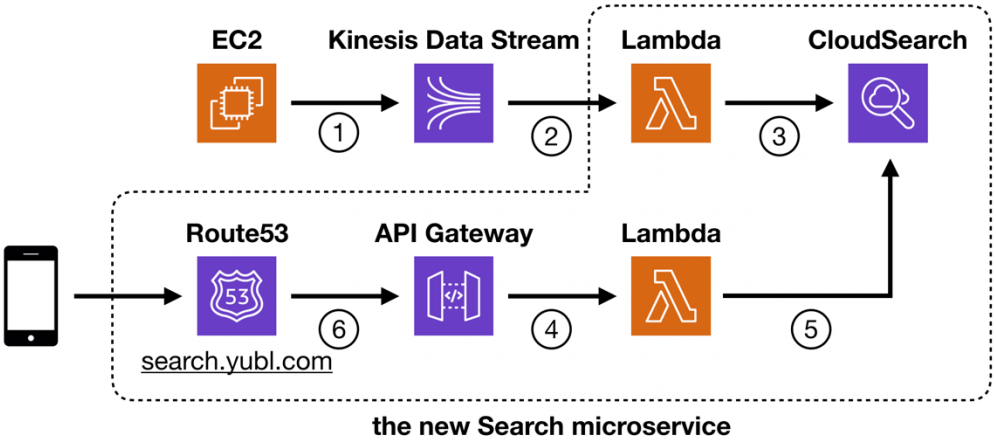


Figure 4.4 A high-level overview of the new architecture running on serverless components.

If you follow the numbered arrows in figure 4.4, this is how all the pieces fit together:

1. The legacy monolith publishes all user-related events to a Kinesis Data Stream called `users`. These include the `user-created`, `user-profile-updated` events that tell us whenever a new user joins or a user has updated his/her profile.
2. A Lambda function is subscribed to the `users` stream.
3. The Lambda function uses these events to insert, update, or delete user documents in the `users` index in Amazon CloudSearch.
4. A new API in API Gateway with a `POST /?query={string}` endpoint proxies to another Lambda function to handle the HTTP request.
5. The Lambda function translates a user's query string into a search request against the `users` index in Amazon CloudSearch.
6. To create a user-friendly subdomain for the new REST API, a custom domain name in API Gateway for `search.yubl.com` is registered in Route53 too.

For this microservice, we chose Amazon CloudSearch instead of Amazon ElasticSearch because at the time Amazon ElasticSearch didn't allow you to change the number of write nodes in an ElasticSearch cluster, which is a scalability concern for the write throughput.

But Amazon CloudSearch was not without its problems. Although you can auto-scale the read and write nodes independently, scaling up a CloudSearch cluster takes as long as 30 minutes. This did not match well with our spiky workload, and we had to over-provision the read cluster as a result. If I implemented this service again today, I would definitely use Amazon ElasticSearch or a third-party service such as Algolia (https://algolia.com) instead.

Before we launched the new service, we also needed to ensure all existing user data was available in the CloudSearch index. To do this, we ran a one-off task to copy all existing user data (~800,000 users) from MongoDB to CloudSearch while tracking the most recent user

profile update. And only after this was complete, did we enable the function at step 2 (figure 4.4) to start processing user updates. Another important detail to note here is that when we enabled the function's Kinesis subscription, we processed events from when the one-off task started. With Kinesis, you are able to specify the StartingPosition of the subscription. You can configure this to `AT_TIMESTAMP` to start processing events from a specific timestamp. Processing events from when a one-off task started ensured that we didn't miss any updates that happened while it was running.

Once live, performance of the new search service was significantly improved over the old search when it was run against a database built for purpose. It also removed a lot of the load on the monolith MongoDB database in the process, which had a positive impact on the general responsiveness of the app. It also gave us a template on how to build other microservices using serverless technologies such as API Gateway and Lambda.

4.3 Migrating to new microservices gracefully

Building the new microservices was the easy part. The difficult part was how to migrate them safely and be able to roll back quickly if there were any unforeseen issues. Another concern was how to do it gracefully without downtime and impact on our users.

Suppose your starting position is a monolith where all the features are accessing a shared, monolith database directly (figure 4.5). Where will you begin?

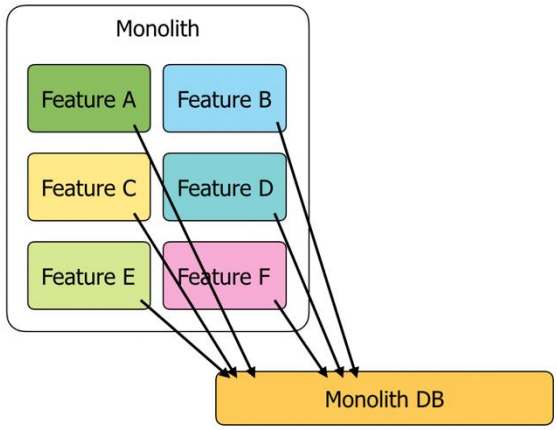


Figure 4.5 A monolithic system where everything has direct access to a shared database.

You start to break apart this monolith into microservices that are built with serverless components such as API Gateway, Lambda and DynamoDB. As you move a feature out of the monolith into its own microservice, you want the microservice to be the authority over some part of the system, be it user profiles or product catalogue or customer orders. The microservice has its own database, and other microservices (or the monolith) should not be able to reach into its database and access or manipulate data directly.

Instead, to instigate some state change, other microservices needs to communicate with this microservice through its API, which can be HTTP-based in the form of a REST API call, or message-based in the form of publishing an event/message to a queue. The important thing is to cut off the direct access to and manipulation of data the microservice is supposed to be the authority of (figure 4.6). But how do you do this gracefully without causing significant disruption to your users?

The challenge here is that it's risky to do a big-bang migration and usually requires downtime. That is not to say that you should never entertain the idea of a big-bang migration. If you're a small startup and have few users on your current platform then it's quite possibly the fastest and most efficient approach for you to roll out these new microservices. But for many organizations that are undergoing such migration, it's important to minimize the risk and disruption caused by moving to a new microservice.

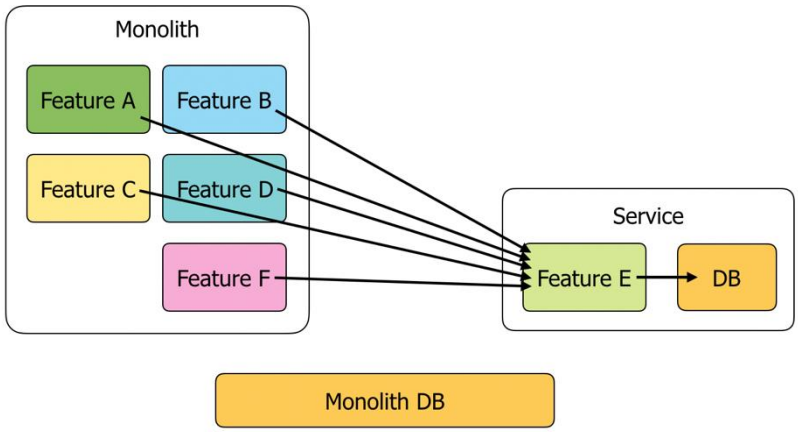


Figure 4.6 A monolithic system where everything has direct access to a shared database

A common strategy is to perform the migration in multiple steps to maximize safety, like those described in the following process:

1. First, move the business logic for a particular feature into a separate service and create its own API. The new service will still use the monolith database until it has authority over the data (figure 4.7).
2. Find the places where the monolith is accessing this feature's data directly and move that access to go through the new service's API instead. Start with the least critical component first to minimize the blast radius of any unforeseen problems or impacts (figure 4.7).
3. Move all other direct access to the new service's data to go through its API, probably one at a time (figure 4.7).
4. Now that the new service is the authority over its data, you can plan a course to migrate the data out of the monolith database into its own database. You might use a different

database to the monolith based on your requirements for this new service. If your access pattern is simple and mostly key lookups, then DynamoDB is probably the right choice (figure 4.7).

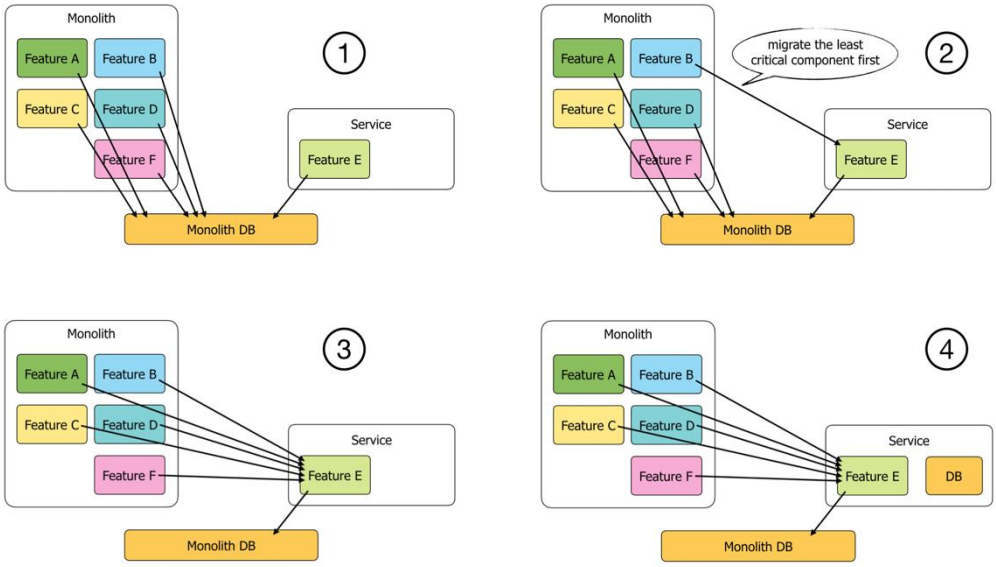


Figure 4.7 Gradually cut off the direct access to the shared monolith database by moving access to go through the new microservice's API instead.

5. Once you have created the new database, you need to migrate data from the monolith database. To do so without downtime, you can treat the new database as a read-through and write-through cache. That is, any updates and inserts are written to the monolith database and then copied to the new database. When attempting to read, you will read from the new database first; if the data is not found, then read from the monolith database and save the data in the new database (figure 4.8).
6. And last, run a one-off task in the background to copy over all existing data. Take care to ensure that you don't overwrite newer updates. With DynamoDB, this can be done using conditional writes (see https://amzn.to/2IbE818) (figure 4.8).

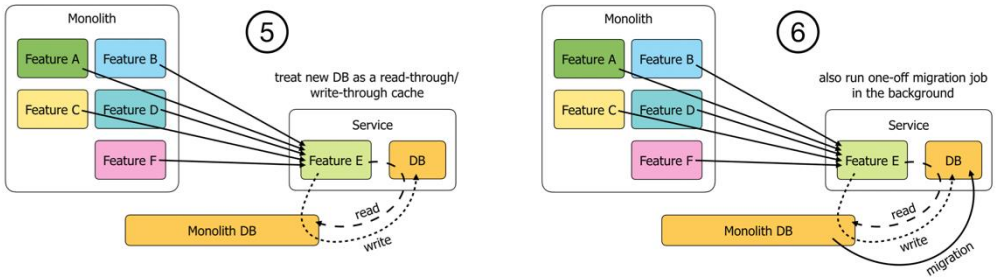


Figure 4.8 Migrate data to the new data gradually without downtime.

This is a useful pattern for extracting features out of a monolith and moving them into microservices that can scale and fail independently. But there is more you can do to ensure you do so safely and gracefully to minimize the potential impact on your users.

For example, you can route only a small percentage of traffic to the new microservice when it first goes live. This limits the blast radius of any unforeseen problems with the new microservice. It is especially important for microservices that are user-facing and handle requests from the mobile/web client directly because they can have a big impact on user experience.

If you're using the Application Load Balancer (ALB) in front of your application already, then you can configure this routing behavior there (figure 4.9).

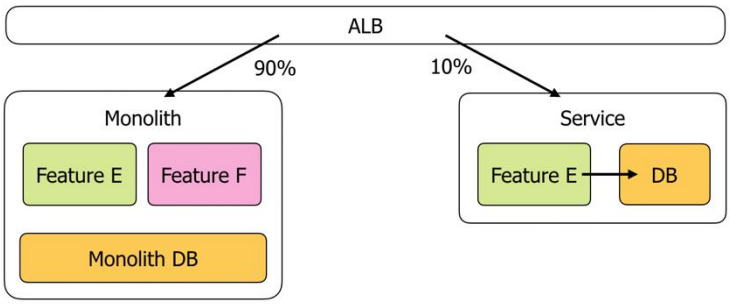


Figure 4.9 You can use the Application Load Balancer (ALB) to distribute traffic between the monolith and the new microservices. This allows you to minimize impact of unforeseen problems to a subset of users.

Where this approach is not possible or in the case of Yubl where ALB didn't exist at the time, you can also proxy requests from the monolith for a configurable percentage of requests (figure 4.10).

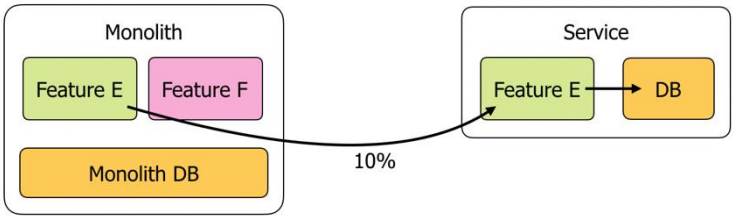


Figure 4.10 Even without ALBs, you can still proxy requests by modifying the monolith.

4.4 Summary

- You need to re-architect most applications to reap the full benefit of a serverless architecture. Though there are solutions to lift-and-shift existing applications into serverless, these don't deliver optimal performance and scalability.
- You need small, autonomous teams who are capable of making their own architectural decisions. Developers should be responsible for more than just the code and empowered to own their system. As Amazon's motto goes, "You build it, you run it." That is how you're going to get the full benefit of a serverless architecture.
- DevOps is simpler with serverless. You get a lot of automation out-of-the-box, and tools such as Serverless framework takes care of the rest. But you still need to know what metrics to pay attention to and what alerts to add. Operational experience of running a production system is still very valuable.
- Unit tests have low return-on-investment when it comes to serverless architectures. Most functions are simple and often integrate with other services such as DynamoDB and Simple Queuing Service (SQS). Unit tests that mocks these integration points do not test those service interactions and give false sense of security. Prefer tests
- Simulating AWS services (for example, DynamoDB, SNS, SQS) locally is not worth the effort. It's easier and quicker to deploy a temporary stack then using local simulation tools.
- When dealing with batched event sources such as Kinesis and SQS, you need to think about how you handle partial failures.

5

A Cloud Guru case study – architecture highlights, lessons learned

This chapter covers

- A Cloud Guru's original REST Architecture
- The reasons the A Cloud Guru's team decided to migrate to Microservices and GraphQL
- Lessons learned through the migration from REST to GraphQL and Microservices

In the first edition of this book, we described a serverless LMS (Learning Management System) built by A Cloud Guru. At that time A Cloud Guru built a RESTful API backend using Amazon API Gateway, AWS Lambda, and Google's Firebase as its primary database. Since we published our first edition, A Cloud Guru has gone through a major transformation. The company moved from a RESTful monolithic design to a GraphQL-driven microservices architecture. This chapter describes this journey. We look at the original RESTful design, the transition to microservices, how GraphQL plays a major part, and the lessons learned along the way.

One thing is clear though, serverless technologies allow you to re-architect big systems quickly. As a developer you are certainly more *agile* with a serverless application than with a traditional three-tier behemoth running on servers. This is because in a serverless approach your primary focus is on the architecture of the system and the code (functions). You don't need to worry about provisioning servers or worry about deploying containers. The A Cloud Guru developers were able to change whole swaths of the system by changing a CloudFormation script and redeploying to AWS. No, nothing is ever just that simple, but the mere fact that they didn't need to think about server capacity or provisioning new or different infrastructure saved a lot of time.

5.1 The original architecture

A Cloud Guru (https://acloud.guru) is an online education platform for anyone wanting to learn Amazon Web Services, Microsoft Azure, Google Cloud Platform as well as Cloud-related technologies. The core features of the platform include the following:

- on-demand video courses
- practice exams and quizzes
- a real-time discussion forum
- dashboards and reporting
- user profiles & gamification
- educational features like learning paths
- interactive sandbox environments for students wanting to test their skills

A Cloud Guru is also an e-commerce platform that allows students to pay for a monthly or yearly subscription and have access to content and features. Training architects who create courses for A Cloud Guru can upload videos directly to S3. These videos are immediately transcoded to a variety of formats and resolutions (1080p, 720p, HLS, and so on) and are made available to students.

Back in 2017/2018, the A Cloud Guru platform used Firebase as its primary database. A nice feature of this database is that it allows client devices (that is, the browser on your computer or phone) to receive updates in near real time without refreshing or polling (Firebase uses web sockets to push updates to all connected devices at the same time). The other main components were API Gateway, and AWS Lambda. Figure 5.1 shows a basic high-level view of what that initial REST (Representational State Transfer) architecture looked like.

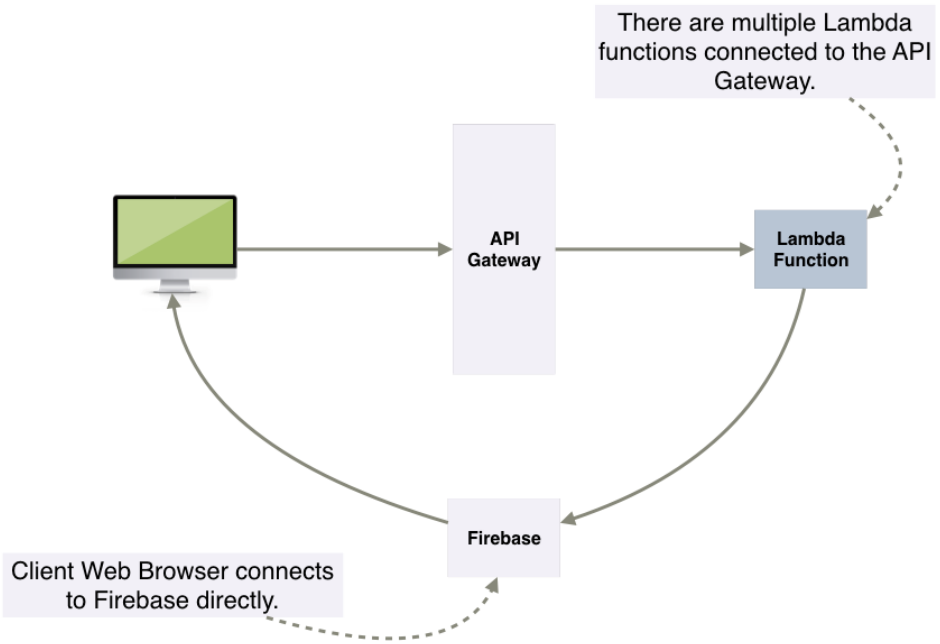


Figure 5.1 A basic, high-level view of the A Cloud Guru architecture. The system was more complex (there were many more Lambda functions) but in a nutshell this is how it worked.

Figure 5.2 shows a slightly more advanced version of the same architecture as was presented in the first edition of this book. In the first edition of our book, we wanted to illustrate that you can build real, sophisticated systems that could supports tens of thousands of users using a REST architecture familiar to a generation of developers. We also wanted to show that you can build sophisticated, scalable, and highly available platforms using functions and services provided by AWS and Google Cloud Platform. The A Cloud Guru team was able to do this, and the system faithfully served tens of thousands of concurrent users.

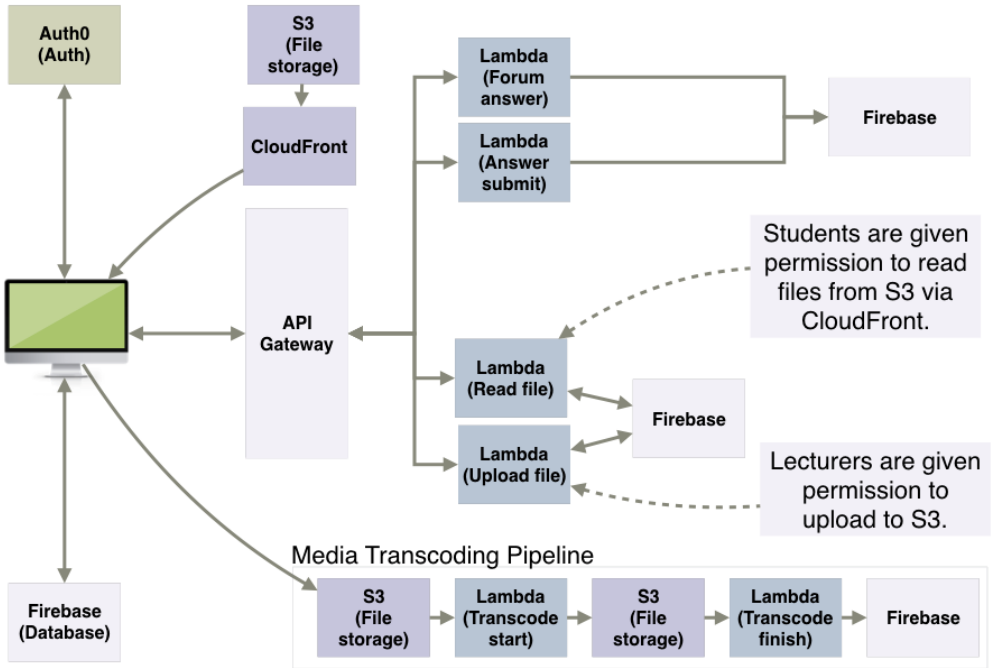


Figure 5.2 This is a slightly more advanced version of the Cloud Guru architecture. The actual production architecture had Lambda functions and services for performing payments, managing administration, gamification, reporting, and analytics.

Note the following about the original A Cloud Guru architecture shown in figure 5.2:

- The front end was built using AngularJS and was hosted by Netlify (https://netlify.com).
- Auth0 is used to provide registration and authentication facilities. It creates delegation tokens that allow the AngularJS website to directly and securely communicate with other services such as Firebase.
- Every client creates a connection to Firebase using web sockets and receives updates from it in near real time. This means that clients receive updates as they happen without having to poll, which leads to a nicer user experience.
- Training Architects who create content for the platform can upload files (usually videos) straight to an S3 bucket via their browser. For this to work, the web application invokes a Lambda function to request the necessary upload credentials first. As soon as credentials are retrieved, the client web application begins a file upload to S3 via HTTP. All of this happens behind the scenes and is invisible to the Training Architect.
- Once a file is uploaded to S3, it automatically kicks off a chain of events that transcodes the video, saves new files in another bucket, updates the database, and immediately makes transcoded videos available to other users.
- To view videos, students are given permission by another Lambda function. Permissions are valid for 24 hours, after which they must be renewed. Files are accessed via

CloudFront. CloudFront ensures that users have low-latency access to videos wherever they may be.

The original system worked well and scaled to many thousands of users. It was also inexpensive to run, with the AWS bill being just a few thousand dollars (Lambda & API Gateway bill was under $1000USD).

Over time, the A Cloud Guru development team began considering the future of their serverless REST architecture. There were a few reasons to begin looking at alternative designs:

- The architecture that was created was in a sense a serverless monolith. There were a large number of Lambda functions, but they connected to the same Firebase database. Making a change to the database would affect nearly every Lambda function and the developers working on them. This made it very easy for developers to step on each other's toes.
- The business wanted to have separate development teams owning different parts of the product. For example, the student-experience team would have to be able to update a database and deploy a Lambda function, without stepping on the toes of the team responsible for billing & reporting.
- Transitioning to a true microservices approach (where each microservice owns its data and its own view of the world) would allow teams to develop the platform in parallel. Each development team would look after a number of microservices and iterate on them as needed. This would mean moving away from a single Firebase database to multiple databases and yet still have a way to read and hydrate data when needed.
- Moving to a microservices approach would give the teams a higher level of isolation. Different subsystems and components within the code base would have clearer boundaries in terms of ownership and a looser coupling.
- The teams wanted a way to minimize round trips to the backend and fetch only the data that was needed. This can be tricky to accomplish with a REST approach, whereas GraphQL makes it easy (as you will soon see).
- Finally, the team wanted to be able to serve multiple clients like mobile and web. Although it's possible and often done with REST, a GraphQL approach makes supporting multiple clients simpler.

Another problem was that Firebase was getting expensive (it was probably the costliest component), so the team wanted to move to a more cost-effective and scalable database. Firebase is a NoSQL, key-value database, so migrating to Amazon's DynamoDB looked like the right thing to do. Moving to DynamoDB would also allow teams to better manage infrastructure using CloudFormation and leverage built-in Dynamo features like event triggers. Finally, it would allow teams to stay entirely within the AWS environment.

Moving to a more proper microservices approach and having DynamoDB act as the primary database for each microservices necessitated a rethink of the entire architecture. One of the main questions to consider was how to get data from disparate microservices and do it as effectively as possible (without multiple round trips or data hydration on the client) when a user made a request. This is where GraphQL entered the picture and became the focus of the new architecture. But before we get to GraphQL let's see how the A Cloud Guru team split up their monolith and created their microservices first.

A Serverless Monolith

The RESTful API design that the A Cloud Guru created was a serverless monolith. There was a single database and functions that needed to save or load data connected to it. There is nothing wrong with building a serverless monolith. For A Cloud Guru, it scaled well for a long time and helped build the company. The core reason for the move to a microservices design was the need for multiple teams to work in parallel. So, if you are starting out today, know that it is OK to go with a monolithic approach. When you need to you can go to microservices but just because it is trendy, don't think that you have to do it from the get-go. And, if you are serverless, moving from a monolith to a microservices design isn't all that painful.

5.1.1 The journey to 43 microservices

Here are some stats of the new GraphQL re-architected A Cloud Guru platform at the start of 2020:

- 240 million Lambda invocations per month (100 per second)
- 180 million API Gateway calls per month (70 per second)
- 90TB of data transferred from CloudFront per month (274MB per second)

The team began to break apart the monolith and move to a microservices architecture during 2018. API Gateway and Lambdas were separated into discrete microservices, each with their own responsibilities and view of the world. In the new world of microservices, each service could be as simple as a single DynamoDB table, a couple of Lambda functions, and an API Gateway. Figure 5.3 shows an example of what a couple of basic microservices could look like.

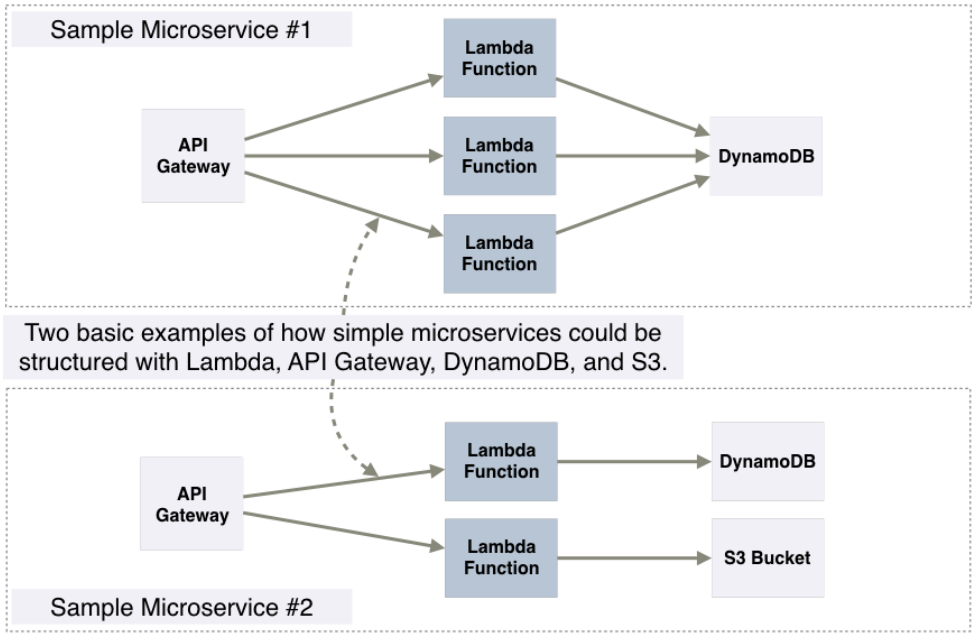


Figure 5.3 The two microservices here are akin to a simplified RESTful architecture we discussed before.

The packaging of the microservices is also interesting to note. A Cloud Guru uses Serverless framework and CloudFormation to organize and deploy microservices. Some services in a microservice are *stateful,* whereas others are *stateless*. A Lambda function is stateless, meaning that it can be overwritten on each deployment. It's always ephemeral. A DynamoDB table or an S3 bucket is stateful; you must be careful to preserve the data that is already there. Your deployment process cannot overwrite what's there. Also, there can be global services and resources that don't belong to any specific microservice. How do you think they should be deployed and managed?

The A Cloud Guru team designed their microservice so they would have different CloudFormation stacks for stateless and stateful resources, as well as a stack for configuration and core dependencies. Figure 5.4 shows what that looks like.

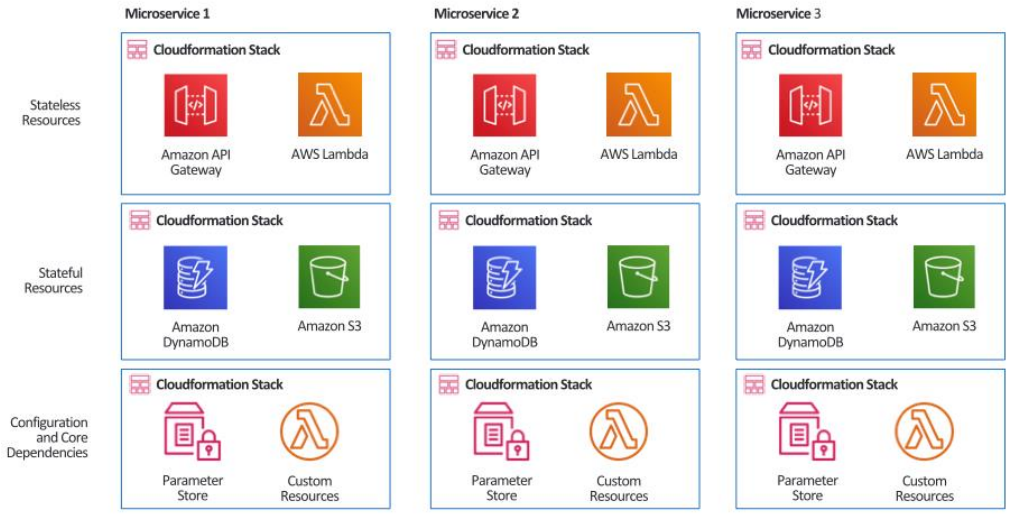


Figure 5.4 Each microservice is in its own CloudFormation stack, which makes it easy to deploy.

The approach to have different CloudFormation stacks for different kinds of resources allows the development team to deploy the stack with the stateless resources when they need to be updated, without having to touch stateful resources. The same goes for the configuration and the core-dependencies stack. They can be updated without modifying anything else within that microservice. This kind of separation of concerns is helpful, because it can help to avoid accidental modification of stateful resources.

There are also a few other global dependencies that exist as well. These are not within any microservice. They include infrastructure components such as Amazon RedShift (data warehouse), AWS WAF (firewall), and VPCs. Microservices have been designed to avoid having a hard dependency on these global resources. In fact, there is quite a loose coupling between them. For example, if a microservice needs to push data into RedShift, it doesn't do it directly. Instead a regular ETL job pulls data out of microservices and writes it to RedShift. That means microservices don't have to know about RedShift. A microservice can live and breathe on its own while a separate ETL task can do its job. Figure 5.5 shows that it's necessary for some resources to live outside specific microservices.

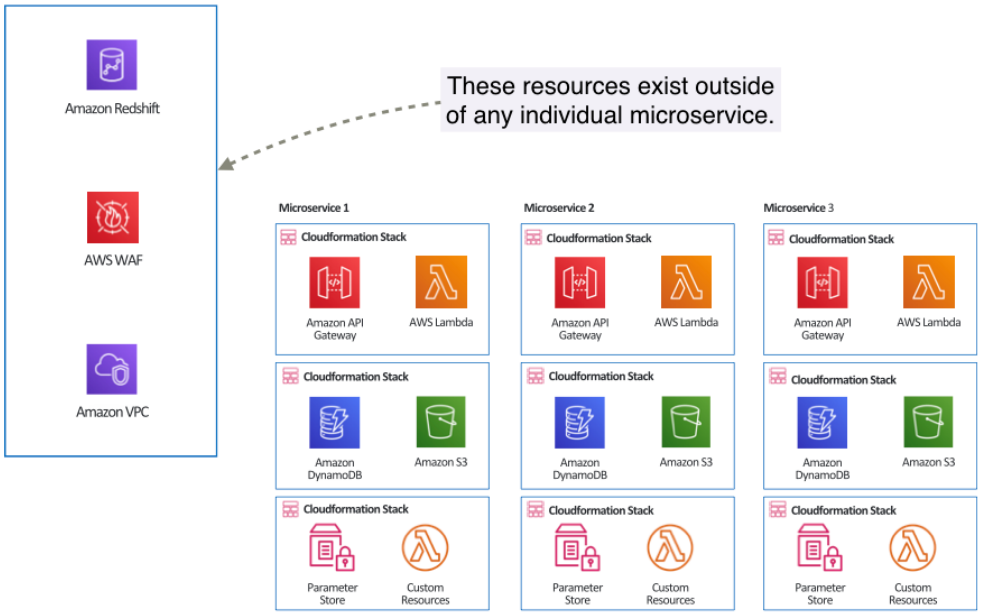


Figure 5.5 There are global dependencies outside of each individual microservice. Not everything has to exist within a microservice.

The A Cloud Guru team gradually teased apart the serverless monolith that was created in the first place and re-implemented it with microservices. Moving from one architecture to a new one always takes time but a nice advantage here was that the team only had to make changes to the serverless.yml, CloudFormation, and the code in various Lambda functions. There was no hardware, servers, or orchestration engine, like Kubernetes, to manage. Also, this re-implementation was able to be done carefully, making sure that no users were affected during the change.

As part of the move to microservices, GraphQL became a solution to a question of how to pull the right data from different microservices when a client makes a request. After all, each microservice may have its own database and its own view of the world. When a user needs to get data, how does it all happen? Which microservice is asked for it? And, what if multiple microservices have the required information and the client needs an aggregate response? GraphQL became the tool to query microservices and, with schema-stitching, create exact responses needed for clients.

5.1.2 What is GraphQL

First, a bit about GraphQL (http://graphql.org) itself. GraphQL is a popular data query language developed by Facebook in 2012 and released publicly in 2015. It was designed as an alternative to REST because of REST's perceived weaknesses (multiple round-trips, over-

fetching, and problems with versioning). GraphQL attempts to solve these problems by providing a hierarchical, declarative way of performing queries from a single end point (for example, api/graphql; figure 5.6).

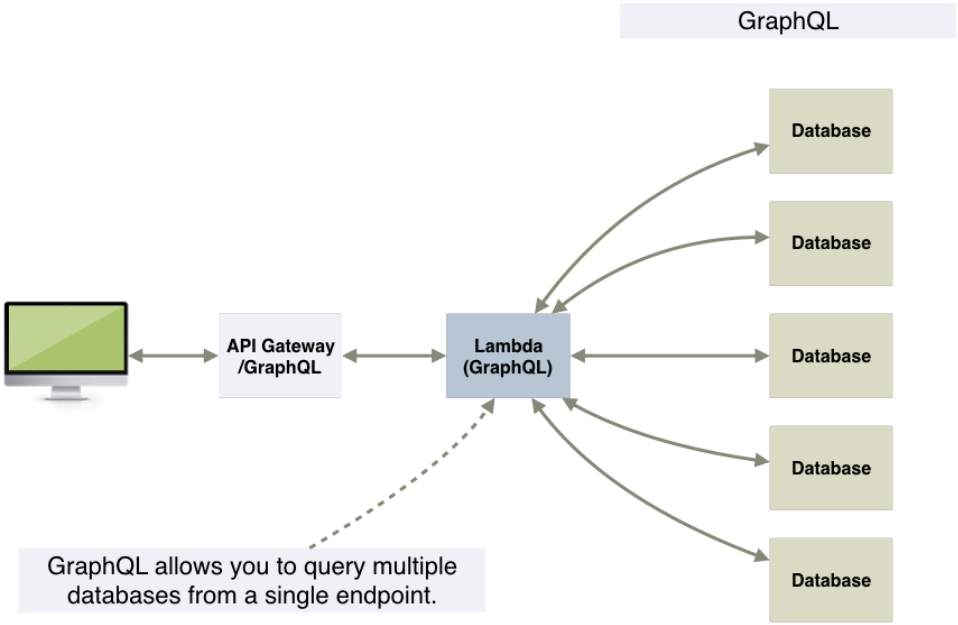


Figure 5.6 A GraphQL library running in a Lambda function can query multiple databases and, using schema stitching, produce a result relevant for the client.

GraphQL gives power to the client. Instead of specifying the structure of the response on the server, it's defined on the client (http://bit.ly/2aTjlh5). The client can specify which properties and relationships to return. GraphQL aggregates data from multiple sources and returns it to the client in a single round trip, which makes it an efficient system for retrieving data. According to Facebook, GraphQL serves millions of requests per second from nearly 1,000 different versions of its application.

In a Serverless architecture, GraphQL can be run from a single Lambda function connected to an API Gateway (this is what A Cloud Guru has done). Or used through a service like AWS AppSync. GraphQL can query and write to multiple data sources, such as DynamoDB tables, and using schema-stitching assemble a response that matches the request.

5.1.3 Moving to GraphQL

When A Cloud Guru began moving to GraphQL, the AWS GraphQL service AppSync wasn't available. So, the team decided to run GraphQL (JavaScript library) from a Lambda function

that is still in use today. The team used the Apollo GraphQL library (https://www.apollographql.com).

> **TIP** Getting the Apollo GraphQL implementation to work in a Lambda function threw in a few interesting quirks and challenges. Apollo was originally designed for long running processes (that is, it was designed to run on servers and containers). The team had to make a certain number of tweaks to make it optimized for Lambda. If the team was implementing a solution now, AppSync would have been looked at closely as a potential alternative.

The A Cloud Guru team began using GraphQL and a design pattern called Backend for Frontend (BFF). The idea behind BFF is that each client has its own API or endpoint that services its specific needs (for example, there's a dedicated endpoint for Mobile and another for Web). Each of the endpoints can query the appropriate microservices to save or load data as needed. The client doesn't need to know about the different microservices. It only needs to know which endpoint to query. This pattern solves the decoupling issue present in many systems. Figure 5.7 shows an example of BFF architecture and what the A Cloud Guru team is driving toward.

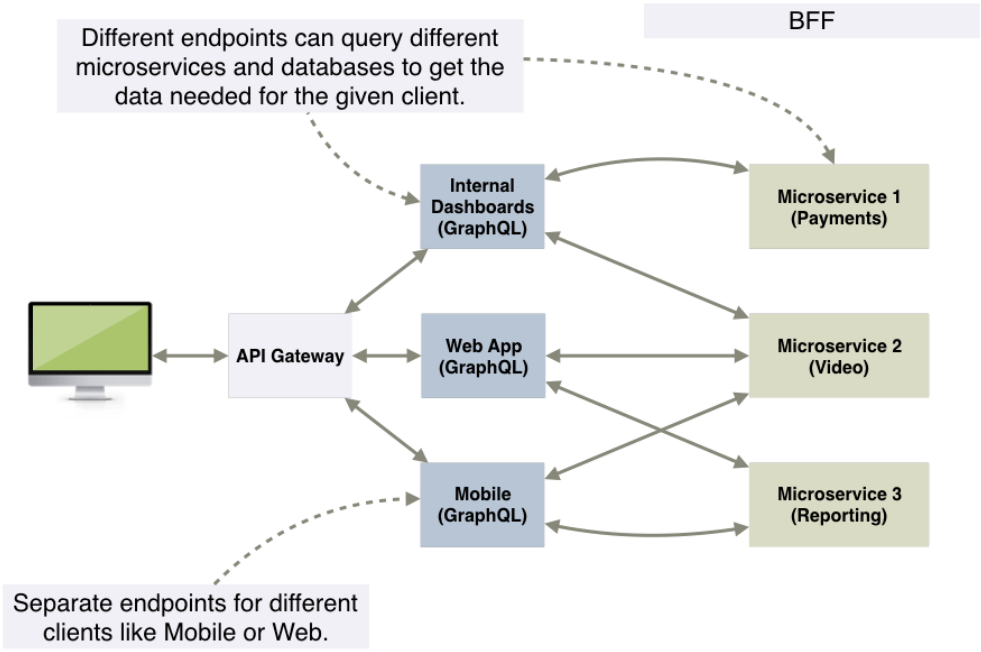


Figure 5.7 An example of the BFF pattern that can be applied to microservices and multiple clients.

Throughout the chapter we've called the Lambda function that contains the GraphQL JavaScript library a GraphQL endpoint. But it's probably better to call it a *BFF endpoint* as we now understand this pattern.

Here's how the A Cloud Guru's implementation works at a very high-level (excluding a few details like service discovery):

- A request from the user reaches the API Gateway.
- API Gateway invokes the Lambda function with the Apollo GraphQL library. This is the BFF endpoint we've discussed.
- The Apollo GraphQL library queries the microservices it knows about (more on how it knows about which microservices to target in a moment – the service discovery section covers it).
- Each microservice has an endpoint which is an API Gateway with a Lambda function (that is, there is a /graphql endpoint in each microservice).
- The Lambda function runs the Apollo GraphQL library with a number of thin schema resolvers. It queries the databases contained within that microservice and produces a result.
- The result is sent back to the BFF endpoint.
- The BFF endpoint gets responses back from microservices and using schema stitching assembles a final response for the client.
- The response is sent back via the API Gateway to the client.

The GraphQL Lambda function is aware of the multitude of microservices (more on this at the moment) and is able to query them when a client request comes through. Figure 5.8 shows a high-level overview of this architecture.

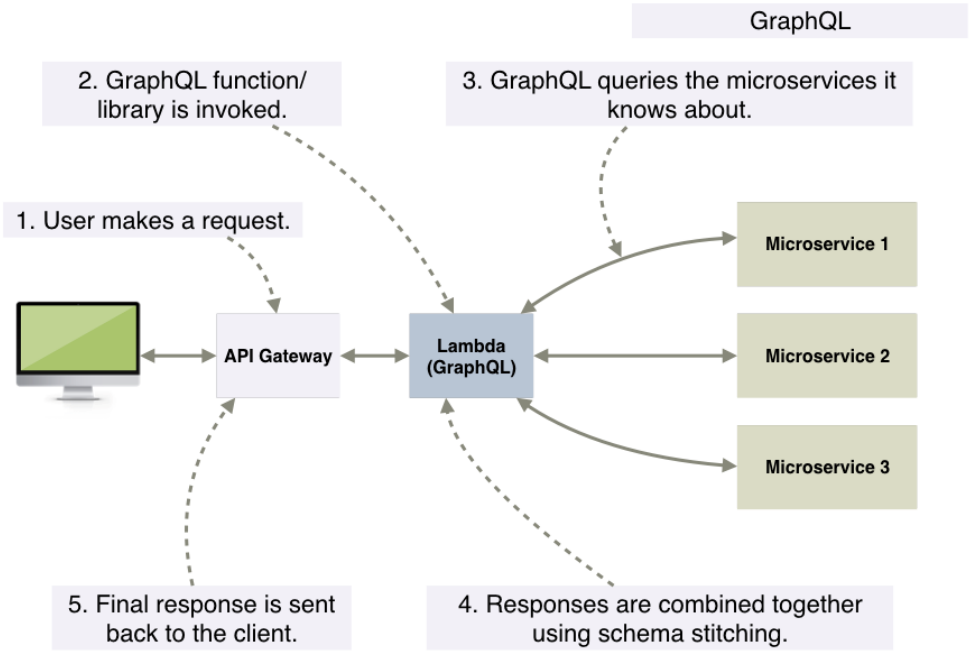


Figure 5.8 The GraphQL endpoint serves as the central point for clients that need data.

5.1.3 Service discovery

Right now, there are 43 microservices in the A Cloud Guru architecture. How does the GraphQL know which services to query when a client request comes in? The team has built an internal service-discovery service called Sputnik (note, this is an in-house, proprietary service that's not available publicly). Sputnik consists of a database with API/URI definitions and database schemas. Microservices know when to update Sputnik, and the GraphQL Lambda function knows when to query Sputnik to get schemas for each microservice and figure out where to route requests (that is, the URI). Service discovery is a standard technique in microservices architecture that solves the problem of knowing what services are available and knowing their interfaces. Sputnik is made up of Lambda functions and DynamoDB tables that contain schemas and URIs of different microservices. It is really a microservice that facilitates the communication of BFF with other microservices in the system. Figure 5.9 shows how Sputnik is used to help the BFF endpoint know where to make a query.

> **TIP:** AWS has a service called Cloud Map which is AWS' own service discovery product. It even has a tagline that simply says, "Service discovery for cloud resources." If you are looking for something like Sputnik, check out Cloud Map. It may work for you. You can find it at https://aws.amazon.com/cloud-map/.

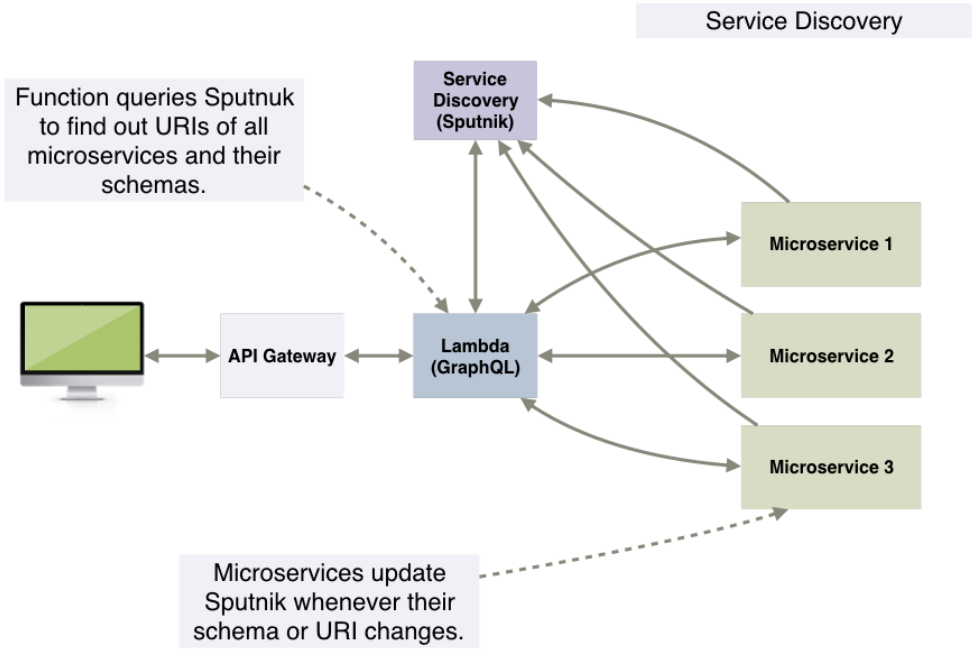


Figure 5.9 The Service Discovery (Sputnik) mechanism. There's now a built in AWS service called Cloud Map you may want to check out.

The metadata about each microservice (URI & schema) is cached at the BFF endpoint too, negating the need for the function to query Sputnik on every request. However, Sputnik can invalidate that cache and force the Lambda function to re-query it again.

5.1.4 Security in the BFF world

The A Cloud Guru practices security in depth and has multiple layers of security built into the system. Let's talk about the two main components: user authentication/authorization and BFF to microservice security.

In the A Cloud Guru platform, students are authenticated using the Auth0 service that generates a unique JWT token for each user. All requests to AWS are made with that JWT token, which is validated using a custom authorizer at the API Gateway. If the JWT token is valid, the request is allowed to continue to the BFF endpoint. If it's not, then a response is generated and sent back to the client telling it that it is unauthorized. This is a simple mechanism that was also used in the original REST design of the platform.

The second interesting element is how to authenticate a request made by the BFF to the microservice. In this scenario A Cloud Guru uses API keys to authenticate requests. Each microservice has a unique API key that the BFF includes in its request in the header (using the X-API-Key parameter). Microservices check the included key and authorize requests if everything is ok.

5.1.5 Remnants of the legacy

The migration from REST to GraphQL took some time because the teams were careful not to cause issues for users. An interesting side-effect of this was the way the system looked like midway through the re-architecture. There were new microservices implemented and a BFF, but the old Firebase database was still in use because it was powering some of the elements of the user interface on the A Cloud Guru website. Figure 5.10 shows what that mid-way architecture looked like.

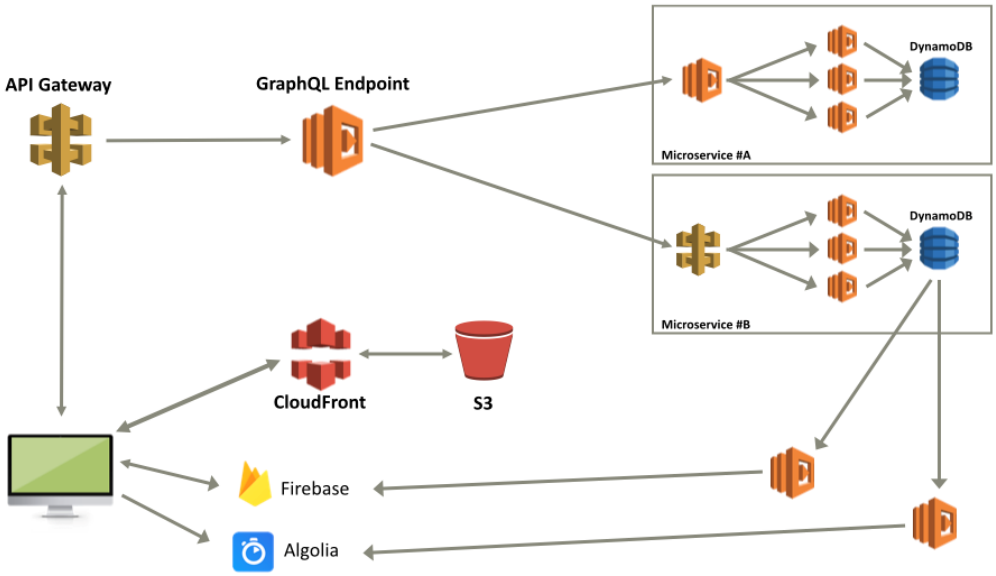


Figure 5.10 A high-level overview of the A Cloud Guru architecture as it was going through a transition.

An interesting note about figure 5.10: you can see that Firebase is still used to drive some of the client-facing user interface. Indeed, there are elements of the UI that remain dependent on Firebase (the A Cloud Guru team should be off it by the time you read this). To keep data in Firebase up to date, DynamoDB event streams and Lambdas are used to make sure it happens. Whenever a table is updated in any microservice, that change is pushed by DynamoDB on to the DynamoDB event stream, which in turn invokes a Lambda function. That Lambda function analyses the change and then updates Firebase (and any other services like Algolia as needed). Now, Firebase becomes basically a materialized view that is used just to drive some parts of the interface. It is never directly queried but it is there for older components that depend on it. This is one of the creative decisions made by the team as they

transitioned from the old serverless architecture to the new one. They were able to use Firebase as they introduced DynamoDB and microservices and move everything across.

There's also an important lesson here in migration. You can gradually implement a new architecture while keeping the old one going by splitting things into smaller pieces and moving them one by one.

5.2 Lessons learned

- Teams can work on the platform without affecting each other. Different teams are responsible for different microservices and they can work on them without affecting anyone else.
- There has been a substantial improvement in performance. The BFF pattern only fetches the data that's needed (great for mobile) and only needs one roundtrip to make it happen. This is an optimization on what was there previously.
- The BFF pattern helps to support multiple client types and devices. They can be different depending on the requirements of the client.
- One of the cons of GraphQL is that there isn't caching like traditional REST (such as e-tag, max-age). The team also had to do additional re-engineering to make Apollo work well with Lambda. These days, it shouldn't be much of a problem but that's the pain when you are an early adopter.
- Security is a number #1 concern. More microservices create a larger surface area for attacks. It's critical that microservices and endpoints are secured. Hence, the use of machine keys to secure communications between backend components is recommended.

7

Building a scheduling service for ad-hoc tasks

This chapter covers

- **Approaching architectural decisions when faced with a novel problem**
- **Defining non-functional requirements**
- **Choosing the right AWS service that can satisfy non-functional requirements**
- **Combining different AWS services to compensate each other's shortcomings**

With serverless technologies, you can build scalable and resilient applications quickly by offloading infrastructure responsibilities to AWS. Doing so allows you to focus on the needs of your customers and your business. Ideally, all the code you write is directly attributed to features that differentiate your business and add value to your customers.

What this means in practice is that you use many managed services instead of building and running your own. For example, instead of running a cluster of RabbitMQ servers on EC2, you use Amazon Simple Queue Service (SQS). Throughout the course of this book, you have also read about other services, such as DynamoDB and Step Functions.

Therefore, an important skill is to be able to analyse the non-functional requirements of a system and choose the correct AWS service to work with. But the AWS ecosystem is enormous and consists of a huge number of different services. Many of these services overlap in their use cases but have different operational constraints and scaling characteristics. For example, to add a queue between two Lambda functions to decouple them, you can use any of the following services:

- Amazon SQS
- Amazon Simple Notification Service (SNS)
- Amazon Kinesis Data Streams
- Amazon DynamoDB Streams

- Amazon EventBridge
- AWS IOT Core

These services have different characteristics when it comes to their scaling behavior, cost, service limits and how they integrate with Lambda. Depending on your requirements, some might be better fit for you than others.

AWS gives you a lot of different services to architect your system, but it doesn't offer any guidance or opinion on when to use which. As a developer or architect working with AWS, one of the most challenging tasks is figuring this out for yourself.

This chapter shines a light on the problem by taking you through the design process for a scheduling service for ad-hoc tasks. It's a common need for applications, and AWS does not yet offer a managed service to solve this problem. The functional requirement for such a scheduling service is simple – it lets you schedule an ad-hoc task to be run at a specified date and time. For example, "remind me to call mum on Monday, at 9:00". What's interesting about it is that it has to deal very different non-functional requirements depending on the application. For example, "it needs to handle a million open tasks that are scheduled but not yet run."

For the rest of this chapter, you will see different solutions for this service using different AWS services and learn how to evaluate them. But first, you need to define these non-functional requirements that we will evaluate the solutions against.

Here is the plan for this chapter:

1. Define non-functional requirements. Here we define the four non-functional requirements we will consider – precision, scalability (number of open tasks), scalability (hotspots) and cost. All the subsequent solutions will be evaluated against these requirements.
2. Cron job with EventBridge. A simple solution using cron jobs to find open tasks and run them.
3. DynamoDB TTL. A creative use of DynamoDB's time-to-live (TTL) mechanism to trigger and run the scheduled ad-hoc tasks.
4. Step Functions. Leverage Step Function's `Wait` state to schedule and run tasks.
5. SQS. A solution that uses SQS's `DelaySeconds` and `VisibilityTimeout` settings to hide tasks until their scheduled execution time.
6. Combining DynamoDB TTL and SQS. A solution that combines DynamoDB TTL with SQS to compensate for each other's shortcomings.
7. Choosing the right solution for your application. Different applications have different needs, and some non-functional requirements may be more important than others. Here, you will see three different applications, understand their needs, and pick the most appropriate solution for them.

7.1 Defining non-functional requirements

The ad-hoc scheduling service is an interesting problem that often shows up in different contexts and has different non-functional requirements, depending on the context. For example, a dating app may require ad-hoc tasks to remind users a date is coming up. A

multiplayer game may need to schedule ad-hoc tasks to start or stop a tournament. A news site might use ad-hoc tasks to cancel expired subscriptions.

User behaviors and traffic patterns differ between these contexts, which in turn create different non-functional requirements the service needs to meet. It's important for you to define these requirements upfront to prevent unconscious biases such as confirmation bias from creeping in. Too often, we subconsciously put more weight behind characteristics that align with our solution, even if they aren't as important to our application. Defining the requirements upfront helps us maintain our objectivity.

For a service that allows you to schedule ad-hoc tasks to be run at a specific time, the following are some non-functional requirements you need to consider:

- **Precision**: how close to the scheduled time is the task run?
- **Scalability** (number of open tasks): can the service support millions of tasks that are scheduled but not yet processed?
- **Scalability** (hotspots): can the service run millions of tasks at the same time?
- **Cost**

Throughout this chapter, you will evaluate five different solutions against this set of non-functional requirements. And remember, there are no wrong answers! The goal of this chapter is to help you hone the skill of thinking through solutions and evaluating them. For the rest of the chapter, you will see five different solutions. Each provides a different approach and utilizes different AWS services. However, every solution uses only serverless components and there is no infrastructure to manage.

The five solutions are:

- Cron job with EventBridge
- DynamoDB TTL
- Step Functions
- SQS
- Combining DynamoDB TTL with SQS

After each solution, you will be asked to score the solution against the aforementioned non-functional requirements. You can compare your scores against the author's and see the author's rationale for his scores.

Let's start with the solution for Cron job with EventBridge

7.2 Cron job with EventBridge

This solution uses a cron job in EventBridge (figure 7.1) to invoke a Lambda function every couple of minutes. With this solution, you will need the following:

- A database (such as DynamoDB) to store all the scheduled tasks, including when they should run.
- An EventBridge schedule (cron) that runs every X minutes.

A Lambda function that reads overdue tasks from the database and run them.

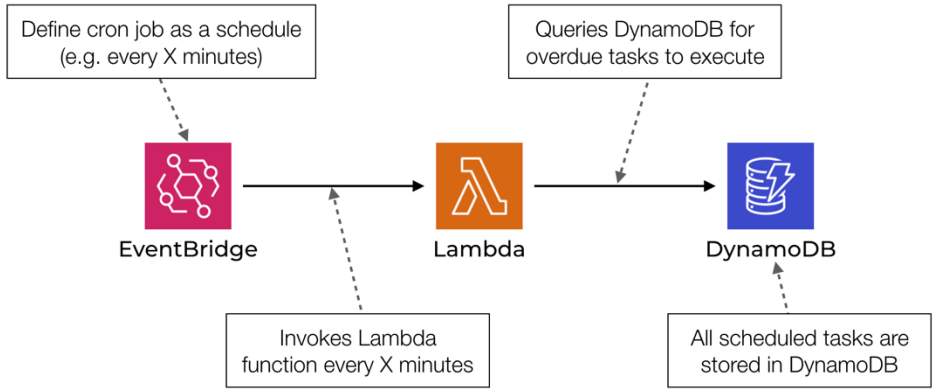


Figure 7.1 High- level architecture of using an EventBridge cron job with Lambda to run ad-hoc scheduled tasks.

There are a few things to note about this solution.

1. The lowest granularity for an EventBridge schedule is 1 minute. So, assuming the service is able to keep up with the rate of scheduled tasks that need to be run, the precision of this solution is "within 1 minute."
2. The Lambda function can run for up to 15 minutes. If the Lambda function fetched more scheduled tasks than it can process in one minute, then it can keep running until it completes the batch. But, in the meantime, another concurrent execution of this function could have been started by the cron job. Therefore, you need to take care to avoid the same scheduled tasks being fetched and run twice.
3. The precision of individual tasks within the batch can vary, depending on their relative position in the batch and when they are actually processed. And in the case of a large batch that cannot be processed within 1 minute, it means the precision for some tasks maybe longer than 1 minute (figure 7.2).

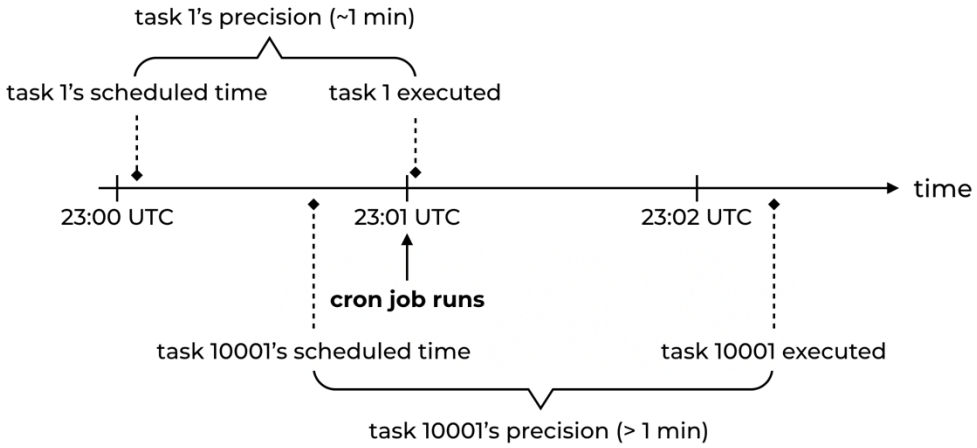


Figure 7.2 The precision of individual tasks inside a batch can vary greatly depending on their position inside the batch.

4. It's possible to increase the throughput of this solution by adding the Lambda function as target multiple times (figure 7.3).

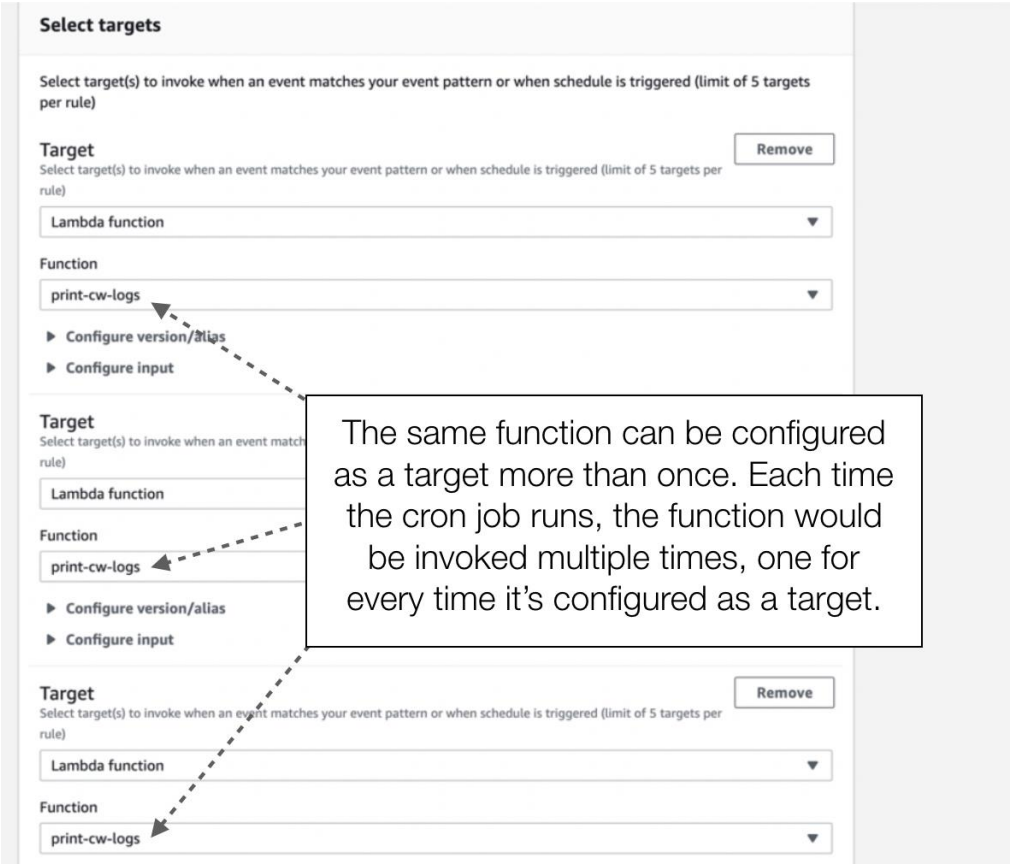


Figure 7.3 You can add the same Lambda function as target for an EventBridge rule multiple times.

5. Because EventBridge has a limit of five targets per rule, you can use this technique to increase the throughput five-fold. This means every time the cron job runs, it will create five concurrent executions of this Lambda function. To avoid them all picking up and running the same tasks, you can configure different inputs for each target, as shown in figure 7.4.

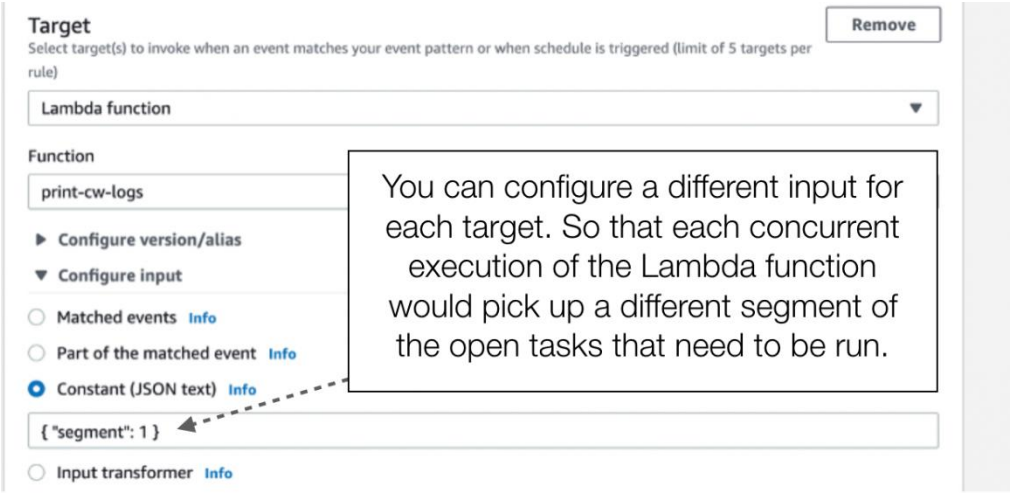


Figure 7.4 You can configure different input for each target to have them fetch different subsets of scheduled tasks, so tasks are not processed multiple times.

7.3 Your scores

What do you think of this solution? How would you rate it on a scale of 1 to 10 against each of the non-function requirements?

Write down your scores in the empty spaces below. And remember, there are no right or wrong answers. Just use your best judgement based on the information available (table 7.2).

Table 7.1 Your solution scores

	Score
Precision	
Scalability (number of open tasks)	
Scalability (hotspots)	
Cost	

7.3.1 My scores

The biggest advantage of this solution is that it's really simple to implement. The complexity of a solution is an important consideration in real-world projects because we're always bound by resource and time constraints. However, for the purpose of this book, we will ignore these real-world constraints and only consider the non-functional requirements outlined in 7.1. With that said, here are my scores for this solution (table 7.1).

Table 7.2 My solution scores

	Score
Precision	6
Scalability (number of open tasks)	10
Scalability (hotspots)	2
Cost	7

Here is how I arrived at these scores.

PRECISION

I gave this solution a 6 for Precision because EventBridge cron jobs can run at most once per minute. Therefore, that's the best precision we can hope for with this solution.

Furthermore, this solution is also constrained by the number of tasks that can be processed in each iteration. When there are too many tasks that need to be run at the same time, they can be stacked up and cause delays. These delays are a symptom of the biggest challenge with this solution – dealing with hotspots. More on that in a minute.

SCALABILITY (NUMBER OF OPEN TASKS)

Provided that the open tasks do not cluster together (hotspots), this solution would have no problem scaling to millions and millions of open tasks. Because each time the Lambda function runs, it only cares about the tasks that are now overdue.

Because of this, I gave this solution a perfect 10 for Scalability (number of open tasks).

SCALABILITY (HOTSPOTS)

I gave this solution a lowly 2 for Scalability (hotspots) because it doesn't handle hotspots well at all. When there are more tasks than the Lambda function can handle in one invocation, this solution runs into all kinds of troubles and forces us into difficult trade-offs.

For example, do we allow the function to run for more than 1 minute?

If we don't, then the function would time out and there's a strong possibility that some tasks might be processed but not marked as so because the invocation was interrupted midway through. We need to either make sure the scheduled tasks are idempotent, or we have to choose between:

1. Executing some tasks twice if we mark them as "`processed`" in the database after successfully processing.
2. Not executing some tasks at all if we mark them as "`processed`" in the database before we finish processing them.
3. Employ a mechanism such as the Saga pattern[1] for managing the transaction and reliably updates the database record after the task was successfully processed. This can add a lot of complexity and cost to the solution.

On the other hand, if we allow the function to run for more than 1 minute then we are less likely to experience this problem. That is, until we see a large enough hotspot that the Lambda function can't process in 15 minutes!

[1] https://theburningmonk.com/2017/07/applying-the-saga-pattern-with-aws-lambda-and-step-functions

Also, now there can be more than 1 concurrent execution of this function running at the same time. To avoid the same task being run more than once, we can set the function's `Reserved Concurrency` to 1. This ensures that at any moment in time, only one concurrent execution of the Lambda function is running (see figure 7.5). However, this severely limits the potential throughput of the system.

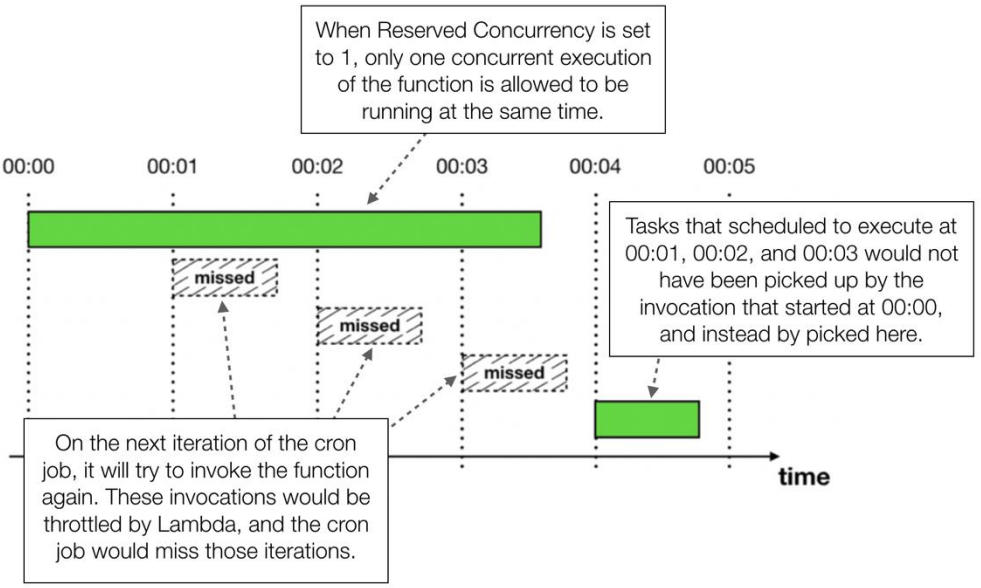


Figure 7.5 If we limit Reserve Concurrency to 1 then there will be only one concurrent execution of the Lambda function running at any moment in time. Which means some cron job cycles will be skipped.

Imagine a million tasks need to be run at 00:00 UTC but the Lambda function can process only 10,000 tasks per minute.

If we do nothing, then the function would timeout, retried, and would take at least 100 invocations to finish all the tasks. In the meantime, other tasks are also delayed, further exasperating the impact on user experience.

This is the Achille's heel of this solution. But we can tweak the solution to increase its throughput and help it cope with hotspots better. More on this later.

Cost

With EventBridge, the cron jobs are free, but we still have to pay for the Lambda invocations even when there are no tasks to run. You can minimize the Lambda cost if you use a moderate memory size for the cron job function. After all, it's not doing anything CPU-intensive and shouldn't need a lot of memory (and therefore CPU).

The main cost for this solution will be the DynamoDB read and write requests. For every task you need one write request (when scheduling the task) and one read request (when the

cron job retrieves it). This access pattern makes it a good fit for DynamoDB's On-Demand pricing and allows the cost of the solution to grow linearly with its scale.

At $1.25 per million write units and $0.25 per million read units, the cost per million scheduled tasks can be as low as $1.50. Of course, that's just the DynamoDB cost, and even that depends on the size of the items you need to store for each task as DynamoDB read/write units are calculated based on payload size. You also have to factor in the Lambda costs too, which also depend on a number of factors such as memory size, and execution duration.

Nonetheless, this is still a very cost-effective solution, even when you scale to millions of scheduled tasks per day. And hence why I gave it a score of 7.

Overall, I think this is a good solution that is also easy to implement. For applications that don't have to deal with hotspots, it can be a very good choice. And as I mentioned earlier, we can also tweak the architecture slightly to address its problem with hotspots.

7.3.2 Tweaking the solution

Earlier, I mentioned that we can increase the throughput of this solution by allowing multiple concurrent executions of the Lambda to run in parallel. As discussed, we can do this by duplicating the Lambda function target in the EventBridge rule.

Because there's a limit of 5 targets per EventBridge rule, we can only hope for a five-fold increase at best. Beyond that, we can also duplicate the EventBridge rule itself as many times as we need.

But even with these tricks, Lambda's 15 minutes execution time limit is still looming over our head. And, we have to shard the reads so that the concurrent executions don't process the same tasks. We also incur higher operational cost and complexity as well. There are more resources to configure and manage, and there are more Lambda invocations and database reads even though most of the time they're not necessary. Essentially, we have provisioned (for lack of a better term) our application for peak throughput all the time.

Increasing throughput this way is ineffective. A much better alternative is to fan-out the processing logic based on the number of tasks that need to run. Lambda's burst capacity limit[2] allows up to 3000 concurrent executions to be created instantly. This allows for a huge potential for parallel processing even if we use just a fraction of it.

For this to work, we need to move the business logic to fetch and run tasks into another Lambda function. From here, we can invoke as many instances of this new function as we deem necessary when faced with a large batch of tasks, see figure 7.6.

[2] https://amzn.to/2BxRuVG

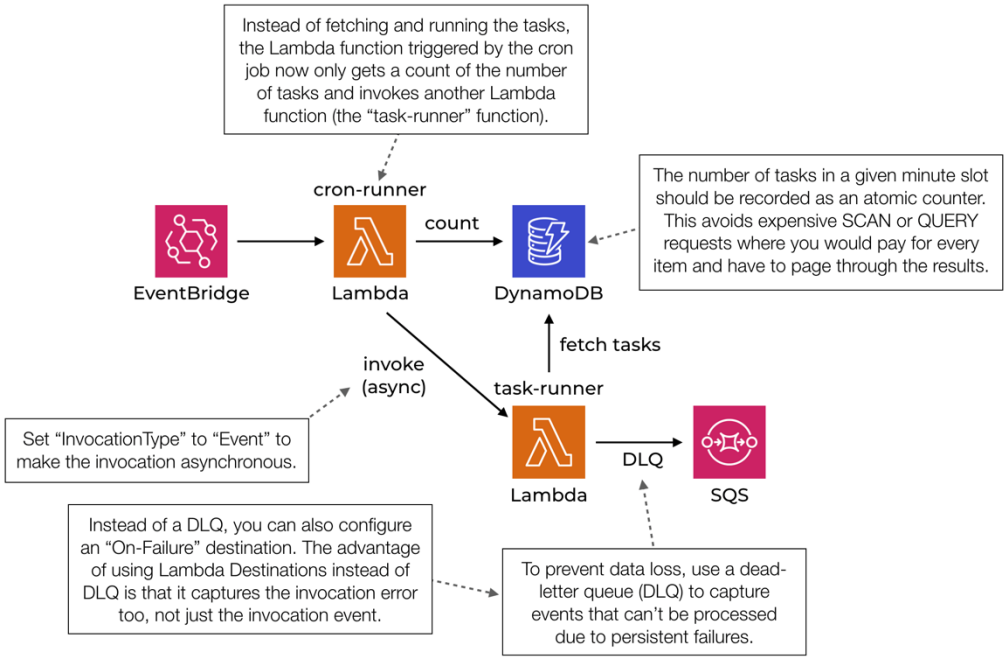


Figure 7.6 An alternative architecture that fans out the processing logic to another function.

Once we know the number of tasks that needs to run, we can calculate the number of concurrent executions we need. To alleviate the time pressure and minimize the danger of timeouts, we can add some headroom into our calculation.

For example, if the throughput for the processing function is 10,000 tasks per minute, then we will start a new concurrent execution for every 5000 tasks. If there are one million tasks, then we need 200 concurrent executions. Which is well below the burst capacity limit of 3000 concurrent executions in the region.

Exercise: score the modified solution

Consider how the proposed changes would affect the non-functional requirements of precision, scalability (number of open tasks), scalability (hotspots) and costs. Write down your score for this modified solution.

Exercise: Other alternatives

While keeping to the same general approach of using cron jobs, are there any modifications to the basic design that can compensate for its shortcomings in precision and scaling for hotspots?

7.3.3 Final thoughts

Cron jobs can be a simple and yet effective solution. And as you can see, with some small tweaks it can also be scaled to support even large hotspots. However, it tends to push a lot of the load onto the database. In the aforementioned scenario of a million tasks that need to be run in a single minute, it will require a million reads from DynamoDB.

Luckily for us, DynamoDB can handle this level of traffic, albeit we need to be careful with the throughput limits that are in place. For example, DynamoDB has a default limit of 40,000[3] read units per table for `On-Demand` tables.

What if there's a way to implement the scheduling service without having to read from the DynamoDB table at all? Well, turns out we can do that by taking advantage of DynamoDB's time-to-live (TTL) feature[4].

7.4 DynamoDB TTL

DynamoDB lets you specify a TTL value on items, and it will delete the items after the TTL has passed. This is a fully managed process, so you don't have to do anything yourself besides specifying a TTL value for each item.

You can therefore use the TTL value to schedule a task that should be run at a specific time. When the item is deleted from the table, a `REMOVE` event will be published to the corresponding DynamoDB Stream. You can subscribe a Lambda function to this stream and run the task when it was removed from the table (see figure 7.7).

[3] https://amzn.to/3eH0THZ
[4] https://amzn.to/2NRgARU

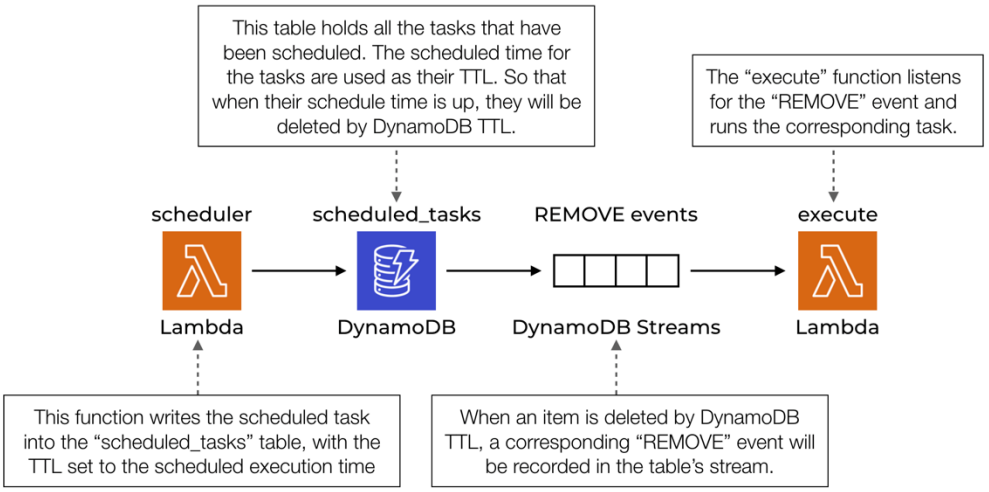


Figure 7.7 High level architecture of using DynamoDB TTL to run ad-hoc scheduled tasks.

There are a couple of things to keep in mind about this solution.

The first and most important is that DynamoDB TTL doesn't offer any guarantee on how quickly expired items are deleted from the table. In fact, the official documentation[5] only goes as far as "TTL typically deletes expired items within 48 hours of expiration", see figure 7.8. In practice, the actual timing is usually not as bleak. Based on empirical data I have collected, items are usually deleted within 30 minutes of expiration. But as figure 7.8 shows, it can vary greatly depending on the size and activity level of the table.

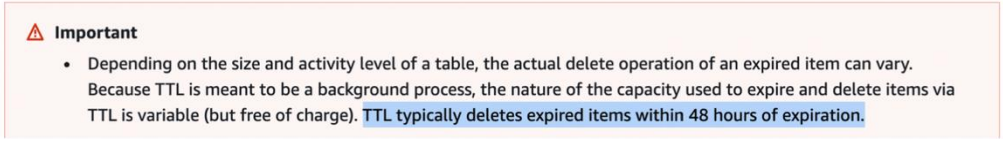


Figure 7.8 "TTL typically deletes expired items within 48 hours of expiration"

The second thing to consider is that the throughput of the DynamoDB Stream is constrained by the number of shards in the stream. The number of shards is in turn determined by the number of partitions in the DynamoDB table. However, there's no way for you to directly control the number of partitions, it's entirely managed by DynamoDB based on the number of items in the table and its read and write throughputs.

Now I know I'm throwing a lot of information at you about DynamoDB, including some of its internal mechanics such as how it partitions data. Don't worry if these are all new to you,

[5] https://amzn.to/2NRgARU

you can learn a lot about how DynamoDB works under the hood by watching this session[6] from re:invent 2018.

7.4.1 Your scores

What do you think of this solution? How would you rate it on a scale of 1 to 10 against each of the non-function requirements?

As before, write down your scores in the empty spaces below.

	Score
Precision	
Scalability (number of open tasks)	
Scalability (hotspots)	
Cost	

7.4.2 My scores

The biggest problem with this solution is that DynamoDB TTL does not delete the scheduled items reliably on time. This limitation means it's not suitable for any application that is remotely time sensitive. With that said, here are my scores.

	Score
Precision	1
Scalability (number of open tasks)	10
Scalability (hotspots)	6
Cost	10

Here is how I arrived at these scores.

PRECISION

Scheduled tasks would be run within 48 hours of their scheduled time... I think 1 might be considered a flattering score here.

SCALABILITY (NUMBER OF OPEN TASKS)

I gave this solution a perfect 10 here because the number of open tasks equals the number of items in the `scheduled_tasks` table. Because DynamoDB has no limit on the total number of items you can have in a table, this solution can scale to millions of open tasks.

Unlike relational databases, whose performance can degrade quickly as the database gets bigger, DynamoDB offers a consistent and fast performance no matter how big it gets.

[6] https://www.youtube.com/watch?v=yvBR71D0nAQ

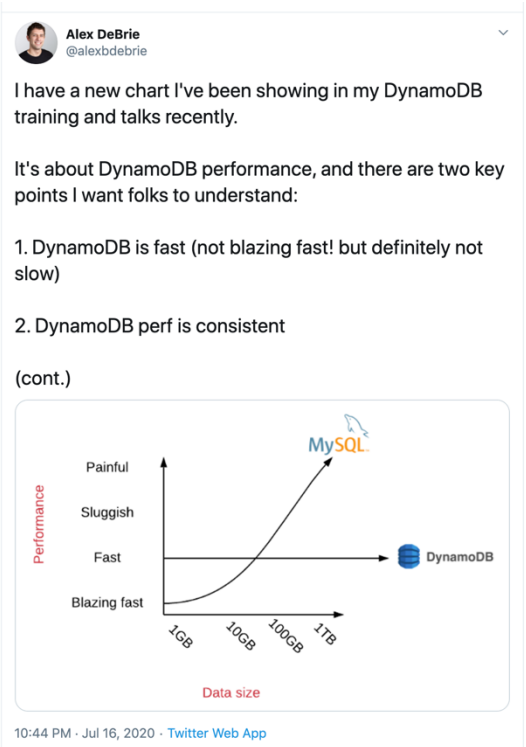


Figure 7.9 "DynamoDB is fast (not blazing fast! But definitely not slow)"

SCALABILITY (HOTSPOTS)

I gave this solution a 6 for Scalability (hotspots) because it can still face throughput-related problems as it's constrained by the throughput of the DynamoDB Stream. But, the tasks would be simply queued up in the stream and be run slightly later than they were scheduled to. Given both of these considerations, I arrived at a score of 6.

Let's drill into this throughput constraint some more as it's useful for you to understand it.

As mentioned earlier, the number of shards in the DynamoDB Stream is managed by DynamoDB. For every shard the Lambda service would have a dedicated concurrent execution of the `execute` function. You can read more about how Lambda works with DynamoDB and Kinesis streams in the official documentation here[7].

When a large number of tasks are deleted from DynamoDB at the same time, the `REMOVE` events will be queued in the DynamoDB Stream for the `execute` function to process. These events would stay in the stream for up to 24 hours. So long the `execute` function is able to eventually catch up then there will not be any data loss.

[7] https://amzn.to/2ZIu3Cx

While there is no scalability concern with hotspots per se, we do need to consider the factors that affect the throughput of this solution:

- How quickly the hotspots are processed depends on how quickly DynamoDB TTL deletes those items. DynamoDB TTL deletes items in batches, and we have no control over how often it runs and how many items are deleted in one batch.
- How quickly the `execute` function is able to process all the tasks in a hotspot is constrained by how many instances of it is running in parallel. Unfortunately, we can't control the number of partitions in the `scheduled_tasks` table, which ultimately determines the number of concurrent executions of the `execute` function. However, we can override the `Concurrent batches per shard`[8] configuration, which allows us to increase the parallelism factor ten-fold (see figure 7.10)!

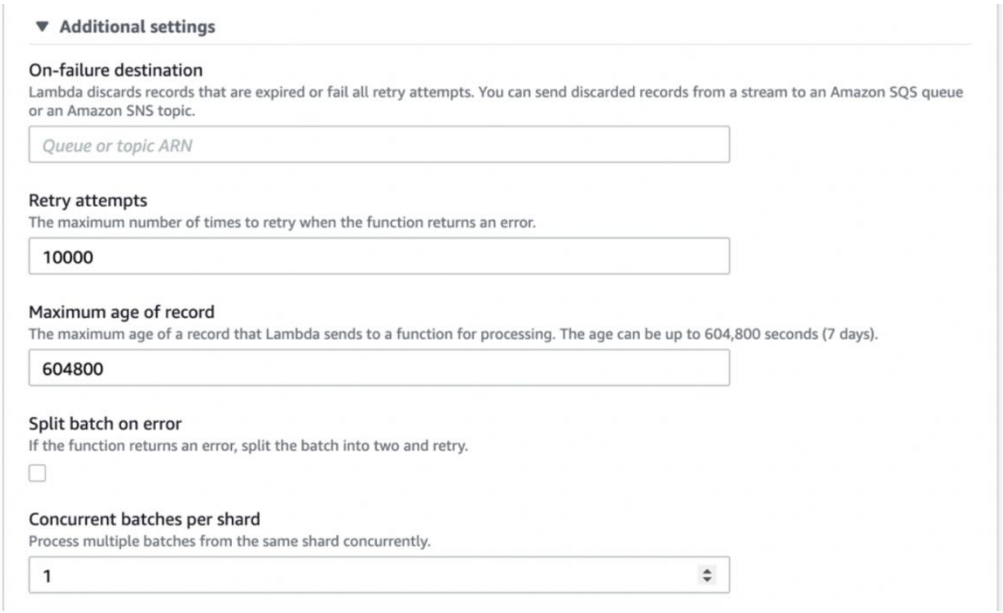


Figure 7.10 You can find the "Concurrent batches per shard" setting under "Additional settings" for Kinesis and DynamoDB Stream functions.

Ultimately, these throughput limitations will affect the precision of this solution.

Cost

This solution requires no DynamoDB reads.

The deleted item will be included in the `REMOVE` events in the DynamoDB Stream. Because events in the DynamoDB Stream are received in batches, they are efficient to process and require fewer Lambda invocations.

[8] https://amzn.to/2YUGE59

Furthermore, DynamoDB Streams are usually charged by the number of read requests, but it's free when you process events with Lambda.

Because of these characteristics, this solution is extremely cost effective even when it's scaled to many millions of scheduled tasks. Hence why I gave it a perfect 10 for Cost.

7.4.3 Final thoughts

This solution makes creative use of the TTL feature in DynamoDB and gives you an extremely cost-effective solution for running scheduled tasks. However, because DynamoDB TTL doesn't offer any reasonable guarantee on how quickly tasks are deleted, it's ill-fitted for many applications.

In fact, neither cron jobs nor DynamoDB TTL are well-suited for applications where tasks need to be run within a few seconds of their scheduled time. For these applications, our next solution might be the best fit as it offers unparalleled precision at the expense of other non-functional requirements.

7.5 Step Functions

Step Functions is an orchestration service that lets you model complex workflows as state machines. It can invoke Lambda functions or integrate directly with a number of other AWS services (such as DynamoDB, SNS and SQS) when the state machine transitions to a new state.

One of the understated superpowers of Step Functions is the `Wait` state[9]. It lets you pause a workflow for up to an entire year! Normally, idle waiting is very difficult to do with Lambda. But with Step Functions, it's as easy as a few lines of JSON:

```
"wait_ten_seconds": {
  "Type": "Wait",
  "Seconds": 10,
  "Next": "NextState"
}
```

You can also wait until a specific UTC timestamp.

```
"wait_until": {
  "Type": "Wait",
  "Timestamp": "2016-03-14T01:59:00Z",
  "Next": "NextState"
}
```

Using `TimestampPath`, you can parameterise the `Timestamp` value using data that is passed into the execution.

```
"wait_until": {
  "Type": "Wait",
  "TimestampPath": "$.scheduledTime",
  "Next": "NextState"
}
```

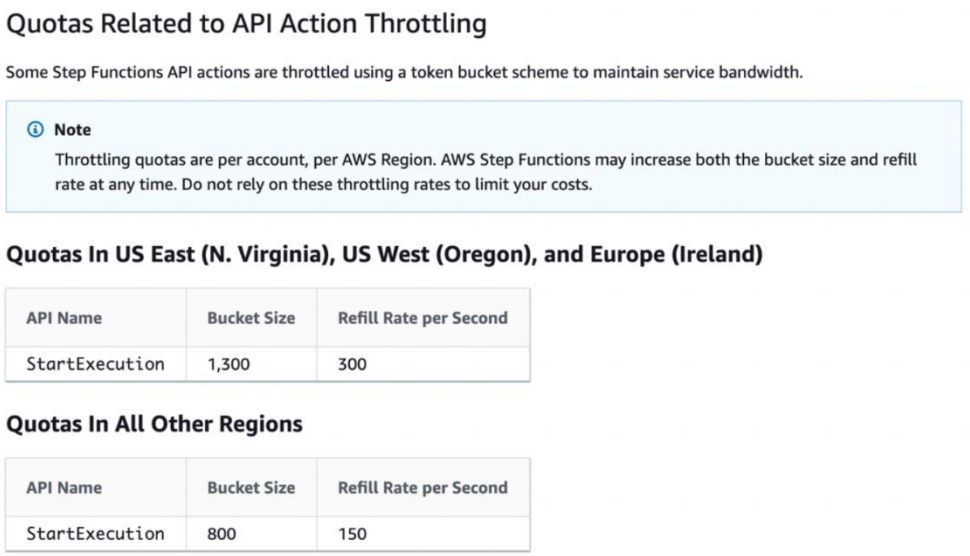
[9] https://amzn.to/38po884

To schedule an ad-hoc task, you can start a state machine execution and use a `Wait` state to pause the workflow until the specified date and time.

This solution is very precise. Based on the data I have collected, tasks are run within 0.01 second of the scheduled time on the 90th percentile.

However, there are several service limits[10] to keep in mind.

- There are limits to the `StartExecution` API, see figure 7.11. It limits the *rate* at which you can schedule new tasks as every task has its own state machine execution.



Quotas Related to API Action Throttling

Some Step Functions API actions are throttled using a token bucket scheme to maintain service bandwidth.

Note

Throttling quotas are per account, per AWS Region. AWS Step Functions may increase both the bucket size and refill rate at any time. Do not rely on these throttling rates to limit your costs.

Quotas In US East (N. Virginia), US West (Oregon), and Europe (Ireland)

API Name	Bucket Size	Refill Rate per Second
StartExecution	1,300	300

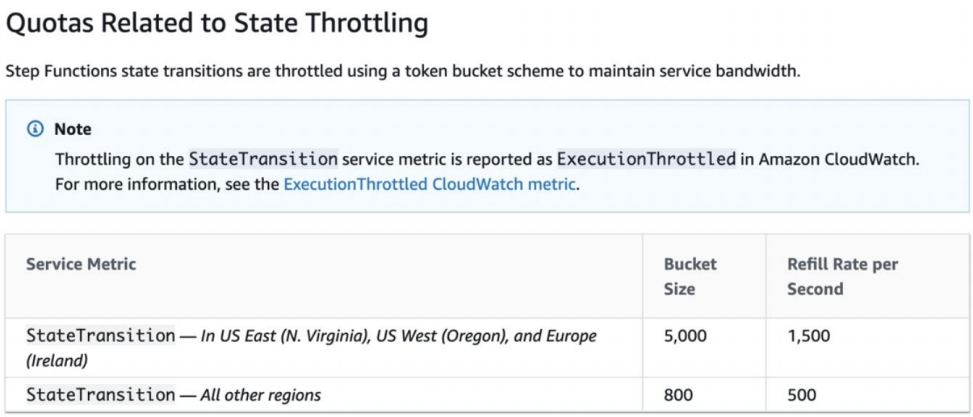
Quotas In All Other Regions

API Name	Bucket Size	Refill Rate per Second
StartExecution	800	150

Figure 7.11 StartExecution API limit

- There are limits to the number of state transitions per second, see figure 7.12. When the `Wait` state expires, the scheduled task is run. However, when there are large hotspots where many tasks all run at the same time, then they could be throttled because of this limit.

[10] https://amzn.to/2C4fGPD



Quotas Related to State Throttling

Step Functions state transitions are throttled using a token bucket scheme to maintain service bandwidth.

Note

Throttling on the `StateTransition` service metric is reported as `ExecutionThrottled` in Amazon CloudWatch. For more information, see the ExecutionThrottled CloudWatch metric.

Service Metric	Bucket Size	Refill Rate per Second
`StateTransition` — *In US East (N. Virginia), US West (Oregon), and Europe (Ireland)*	5,000	1,500
`StateTransition` — *All other regions*	800	500

Figure 7.12 State transition limit

- There is a default limit of 1,000,000 open executions. Because there is one open execution per scheduled task, this is the max number of open tasks the system can support.

Thankfully, all of these limits are **soft limits**. Which means you can increase them with a service limit raise. However, given the default limits for some of these are pretty low, it might not be possible to raise to a level that can support running a million scheduled tasks in a single hotspot.

But there is also the **hard limit** on how long an execution can run one – one year. This limits the system to schedule tasks that are no further than a year away. For most use cases, this would likely be sufficient. If not, we can tweak the solution to support tasks that are scheduled for more than a year away, more on this later.

7.5.1 Your scores

What do you think of this solution? How would you rate it on a scale of 1 to 10 against each of the non-function requirements?

As before, write down your scores in the empty spaces below.

	Score
Precision	
Scalability (number of open tasks)	
Scalability (hotspots)	
Cost	

7.5.2 My scores

Step Functions gives us a simple and elegant solution for the problem at hand. However, it's hampered by several service limits which makes it difficult to scale.

We will dive into these limitations, but first, here are my scores for this solution.

	Score
Precision	10
Scalability (number of open tasks)	7
Scalability (hotspots)	4
Cost	2

PRECISION

As I mentioned before, Step Functions is able to run tasks within 0.01s precision at the 90th percentile. It just doesn't get more precise than that!

SCALABILITY (NUMBER OF OPEN TASKS)

I gave this solution a 7 for Scalability (number of open tasks) because the `StartExecution` API limit restricts how many scheduled tasks we can create per second.

Whereas solutions that store scheduled tasks in DynamoDB can easily scale to scheduling tens of thousands of tasks per second, here we have to content with a default refill rate of just 300/s in the larger AWS regions. Luckily, it is a soft limit, so technically we can raise it to whatever we need. But the onus is on us to constantly monitor its usage against the current limit to prevent us being throttled.

The same applies to the limit on the number of open executions. While the default limit of 1,000,000 is generous, we still need to keep an eye on the usage level as once we reached the limit no new tasks can be scheduled until existing tasks are run. User behavior would have a big impact here. The more uniformly the tasks are distributed over time the less likely this limit would be an issue.

SCALABILITY (HOTSPOTS)

I gave this solution a 4 for Scalability (hotspots) because the limit on `StateTransition` per second is very problematic if a large cluster of tasks need to run around the same time. Because the limit applies to all state transitions, even the initial `Wait` states could be throttled and affect our ability to schedule new tasks.

We can increase both the bucket size (think of it as the bust limit) as well as the refill rate per second. But raising these limits alone might not be enough to scale this solution to support large hotspots with say, a million tasks.

Thankfully, there are tweaks we can make to this solution to help it handle large hotspots better, but we need to trade off some precision. More on this later.

COST

I gave this solution a lowly 2 for Cost because Step Functions is one of the most expensive services on AWS.

We are charged based on the number of state transitions. For a state machine that waits until the scheduled time and runs the task, there are 4 states (see figure 7.13) and every execution would charge for 4 state transitions[11].

[11] https://twitter.com/theburningmonk/status/1279539843039707138

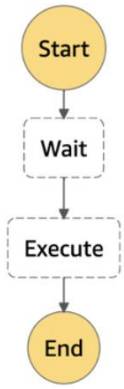


Figure 7.13 A simple state machine that waits until the scheduled time and runs the task.

At $0.025 per 1,000 state transitions, the cost for scheduling 1,000,000 tasks would be $100 plus the Lambda cost associated with executing the tasks. This is nearly two orders of magnitude higher than the other solutions considered so far.

7.5.3 Tweaking the solution

So far, we have discussed several problems with this solution, such as not being able to schedule tasks for more than a year, or having trouble with hotspots. Fortunately, there are simple modifications we can make to address these problems.

EXTENDING THE SCHEDULED TIME BEYOND 1 YEAR

The maximum time a state machine execution can run for is 1 year. As such, the maximum amount of time a `Wait` state can wait for is also a year. However, we can extend this 1-year limitation by borrowing the idea of tail recursion[12] from functional programming.

Essentially, at the end of a `Wait` state we can check if we need to wait for even more time. If so, the state machine will start another execution of itself and wait for another year, and so on. Until eventually we arrive at the task's scheduled time and runs the task.

This is similar to a tail recursion because the first execution does not need to wait for the recursion to finish. It simply starts the second execution and then proceed to complete itself. See figure 7.14 for how this revised state machine might look.

[12] https://en.wikipedia.org/wiki/Tail_call

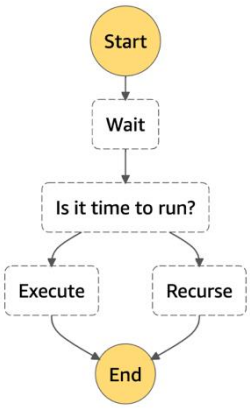


Figure 7.14 A revised sate machine design that can support scheduled tasks that are more than 1 year away.

SCALING FOR HOTSPOTS

Sometimes, just raising the soft limits on the number of `StateTransition` per second alone is not going to be enough. Because the default limits has a bucket size of 5,000 (the initial burst limit) and a refill rate of 1,500 per second.

If we are to support running a million tasks around the same time, we will need to raise these limits by multiple orders of magnitude. AWS will be unlikely to oblige such a request, and we will be politely reminded that Step Functions are not designed for such use cases.

Fortunately, we can make small tweaks to the solution to make it far more scalable when it comes to dealing with hotspots. Unfortunately, we will need to trade off some of the precision of this solution for the newfound scalability.

Add random delay to the scheduled time

For example, instead of running every task scheduled for 00:00 UTC at exactly 00:00 UTC, we can spread the them across a single minute window. We can do this by adding some random delay to the scheduled time. Following this simple change, some of the aforementioned tasks would be run at 00:00:12 UTC, and some would be run at 00:00:47 UTC, for instance.

This allows us to make the most of the available throughput. With the default limit of 5,000 bucket size and refill rate of 1,500 per second, the maximum number of state transitions per minute is 93500:

- Use up all 5,000 state transitions in the first second
- Use up the 1,500 refill per second for the remaining 59 seconds

Doing this would reduce the precision to "run within a minute", but we wouldn't need to raise the default limits by nearly as much. And it'll be a trivial change to inject a variable amount of delay (0-59s) to the scheduled time so that tasks are uniformly distributed across the minute window.

With this simple tweak, Step Functions will no longer be the scalability bottleneck. Instead, we will need to worry about the rate limits on the Lambda function that will run the task.

Run tasks in batches, in parallel

Another alternative would be to have each state machine execution run all the tasks that are scheduled for the same minute.

For example:

- When scheduling a task, add the task in a DynamoDB table with the scheduled time as the HASH key and a unique task ID as the RANGE key. At the same time, atomically increment a counter for the number of tasks scheduled for this timestamp. Both of these updates can be performed in a single DynamoDB transaction. See figure 7.15 for how the table might look like using the single-table design.

timestamp	sk	count	task
2020-07-04T21:53:22	item_count	2	
2020-07-04T21:53:22	item_3013a975-163e-45f6-89c0-357c161bb76c		{ "message": "world" }
2020-07-04T21:53:22	item_3aa4e344-5faf-4fd7-934a-c11f653e5730		{ "message": "hello" }

Figure 7.15 Record the scheduled task as well as count in the same DynamoDB table using single table design.

- Start a state machine execution with the timestamp as execution name. Because execution names have to be unique, the `StartExecution` request will fail if there's an existing execution already.
- Instead of executing the scheduled task immediately after the `Wait` state, get a count of the number of tasks that need to run. From here, use a `Map` state to dynamically generate parallel branches to run these tasks in parallel. See figure 7.16 for how this alternative design might look.

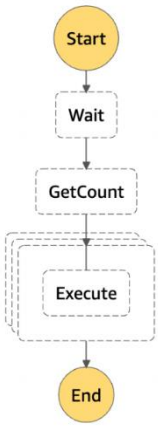


Figure 7.16 An alternative design for the state machine that can run tasks in batches in parallel.

Making these changes would not affect the precision by too much but it would reduce the number of state machine executions and Lambda invocations required. Essentially, we need one state machine execution for every minute when we need to execute some scheduled tasks. There is a total of 525600 minutes in a 365 days calendar year, so this also removes the need to increase the limit on the number of open executions (again, the default limit is 1,000,000).

And that's the beauty of these composable architecture components! There are so many ways to compose them and it gives you lots different options and trade-offs.

Exercise: Score the modified solutions

Repeat the same scoring exercise against the modified solutions I proposed. How much would it impact the system's ability to handle large number of open tasks or hotspots?

Are there any additional cost impact (e.g. one of the proposed tweaks uses a DynamoDB table) that needs to be considered?

7.5.4 Final thoughts

Step Functions offers a single and elegant solution that can run tasks at great precision. The big drawback are its costs and the various service limits that you need to look out for, which hampers its scalability.

But as you can see, if we are willing to make tradeoffs against precision, we can modify the solution to make it much more scalable. We looked at a couple of possible modifications, including taking some elements of the cron job solution and turning this solution into a more flexible cron job that only runs when there are tasks that need to run.

We also looked at a modification that allows us to work around the 1-year limit by applying tail recursion to the state machine design.

In the next solution, we have to apply the same technique as SQS is bound by an even tighter constraint on how long a task can stay open for.

7.6 SQS

The Amazon Simple Queue Service (SQS) is a fully managed queuing service. You can send messages to the queue and receive messages from the queue. Once a message has been received by a consumer, the message is then hidden from all other consumers for a period of time, known as the `Visibility Timeout`. You can configure the `Visibility Timeout` value on the queue, but the setting can also be overridden for individual messages.

When you send a message to SQS, you can also use the `DelaySeconds` attribute to make the message become visible at the right time. You can implement the scheduling service by using these two settings to hide a message until its scheduled time.

However, the maximum `DelaySeconds` is a measly 15 minutes and the maximum `Visibility Timeout` is only 12 hours as well.

But all is not lost. When the `execute` function (see figure 7.17) receives the message after the initial `DelaySeconds`, it can inspect the message and see if it's time to run the task yet. If

not, it can call `ChangeMessageVisibility`[13] on the message to hide the message for up to another 12 hours. It can do this repeatedly until it's finally time to run the scheduled task.

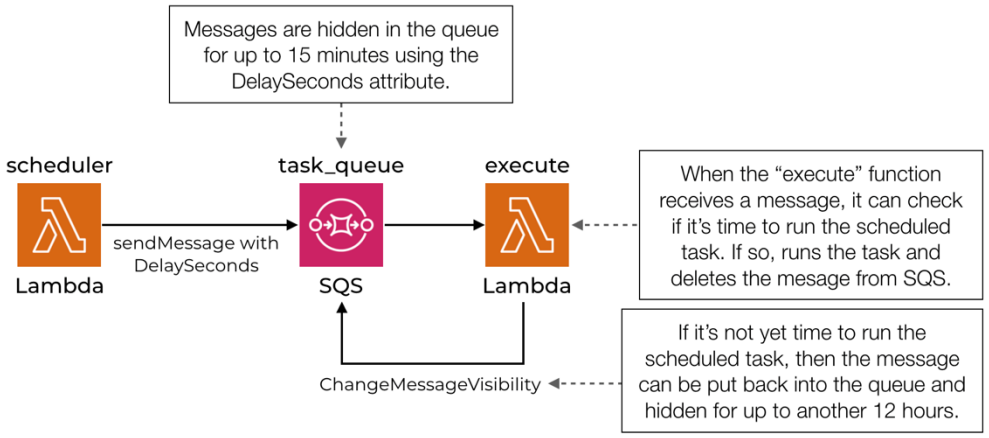


Figure 7.17 High level architecture of using SQS to schedule ad-hoc tasks.

Before you score this solution, consider that there is a limit of 120,000 inflight messages. Unfortunately, this is a hard limit and cannot be raised. This limit has a profound implication that can mean it's not suitable for some use cases at all.

Once a message is inflight, this solution would keep it inflight by continuously extending its `Visibility Timeout` until its scheduled time. In this case, the number of inflight messages equates to the number of open tasks. However, once you reach the 120,000 inflight messages limit then newer messages would stay in the queue's backlog. Even if some of the newer messages might need to be run sooner than messages that are already inflight. Priority is given to tasks based on when they were scheduled, not by their execution time.

This is not a desirable characteristic for a scheduling service. In fact, it's the opposite of what we want. Tasks that are scheduled to execute soon should be given the priority to ensure they're executed on time. That being said, it's a problem that would only arise when you have reached the 120,000 inflight messages limit. So the further away tasks can be scheduled, the more open tasks you would have and the more likely you will run into this problem.

7.6.1 Your scores

With this severe limitation in mind, how would you score this solution? write down your scores in the empty spaces below.

	Score
Precision	

[13] https://amzn.to/3e1GVY6

Scalability (number of open tasks)
Scalability (hotspots)
Cost

7.6.2 My scores

This solution is best suited for scenarios where tasks are not scheduled too far away in time. Otherwise, we face the prospect of accumulating a large number of open tasks and running into the limit on inflight messages.

Also, we would need to call `ChangeMessageVisibility` on the message every 12 hours for a long time. If a task is scheduled to execute in a year, then that's a total of 730 times. Multiplied that by a million tasks and that's a total of 730 million API requests, or $292 for keeping a million tasks open for a whole year.

With these in mind, here are my scores.

	Score
Precision	9
Scalability (number of open tasks)	2
Scalability (hotspots)	8
Cost	5

PRECISION

Under normal conditions, SQS messages that are delayed or hidden are run no more than a few seconds after their scheduled times. Not as precise as Step Functions, but still very good, hence why I gave this solution a score of 9 for Precision.

SCALABILITY (NUMBER OF OPEN TASKS)

I gave this solution a very low score here because the hard limit of 120,000 inflight messages severely limits this solution's ability to support a large number of open tasks. Even though the tasks can still be scheduled, they cannot be run until the number of inflight messages drops below 120,000. This is a serious hinderance and in the worst cases can render the system completely unusable.

For example, if 120,000 tasks are scheduled to run in 1 year, then nothing else that are scheduled after that can be run, until those first 120,000 tasks have been run.

SCALABILITY (HOTSPOTS)

The Lambda service uses long polling to poll SQS queues[14] and only invokes our function when there are messages. These pollers are an invisible layer between SQS and our function and we do not pay for them. But we do pay for the SQS `ReceiveMessage` requests they make.

According to this blog post[15] by Randall Hunt:

[14] https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html#events-sqs-scaling
[15] https://amzn.to/31MfVtI

> "The Lambda service monitors the number of inflight messages, and when it detects that this number is trending up, it will increase the polling frequency by 20 ReceiveMessage requests per minute and the function concurrency by 60 calls per minute. As long as the queue remains busy it will continue to scale until it hits the function concurrency limits. As the number of inflight messages trends down Lambda will reduce the polling frequency by 10 ReceiveMessage requests per minute and decrease the concurrency used to invoke our function by 30 calls per-minute."

By keeping the queue artificially busy with a high number of inflight messages, we are artificially raising Lambda's polling frequency and function concurrency.

This is very useful for dealing with hotspots. Because of the way this solution works, all open tasks are kept as inflight messages. Which means the Lambda service would likely be running a high number of concurrent pollers all the time. So when a cluster of messages become available at the same time, they will likely be processed by the `execute` function with a high degree of parallelism. And Lambda will scale up the number of concurrent executions as more messages become available. We can therefore leverage the auto-scaling capability that Lambda offers.

Because of this, I gave this solution a really good score for Scalability (hotspots).

But on the other hand, this behavior generates a lot of redundant SQS `ReceiveMessage` requests, which can have a noticeable impact on cost when running at scale.

Cost

Between the many `ReceiveMessage` requests Lambda makes on our behalf and the cost of calling `ChangeMessageVisibility` on every message every 12 hours, most of the cost for this solution will likely be attributed to SQS.

While SQS is not an expensive service, but at $0.40 per million API requests, the cost can accumulate quickly as this solution is capable for generating many millions of requests at scale. As such, I gave this solution a 5 for Cost, which is to say that it's not great but also unlikely to cause you too much trouble.

7.6.3 Final thoughts

If you put the scores for this solution side-by-side with DynamoDB TTL then you can see that they perfectly complement each other. Where one is strong, the other is weak.

	Score (SQS)	Score (DynamoDB TTL)
Precision	9	1
Scalability (number of open tasks)	2	10
Scalability (hotspots)	8	6
Cost	5	10

So, what if we can combine these two solutions to create a solution that offers the best of both worlds?

7.7 Combining DynamoDB TTL with SQS

So far, we have seen that the DynamoDB TTL solution is great at dealing with a large number of open tasks, but lacks the precision required for most use cases. Conversely, the SQS solution is great at providing good precision and dealing with hotspots but can't handle a large number of open tasks. The two rather complimentary of each other and can be combined to great effect.

For example, what if long-term tasks are stored in DynamoDB until two days before their scheduled time? Why two days? Because it's the only soft guarantee that DynamoDB TTL gives – "TTL typically deletes expired items within 48 hours of expiration"[16].

Once the tasks are deleted from the DynamoDB table, they are moved to SQS where they are kept inflight until the scheduled time (using the `ChangeMessageVisibility` API as discussed earlier). For tasks that are scheduled to execute in less than two days, they are added to the SQS queue straight away.

See figure 7.18 for how this solution might look.

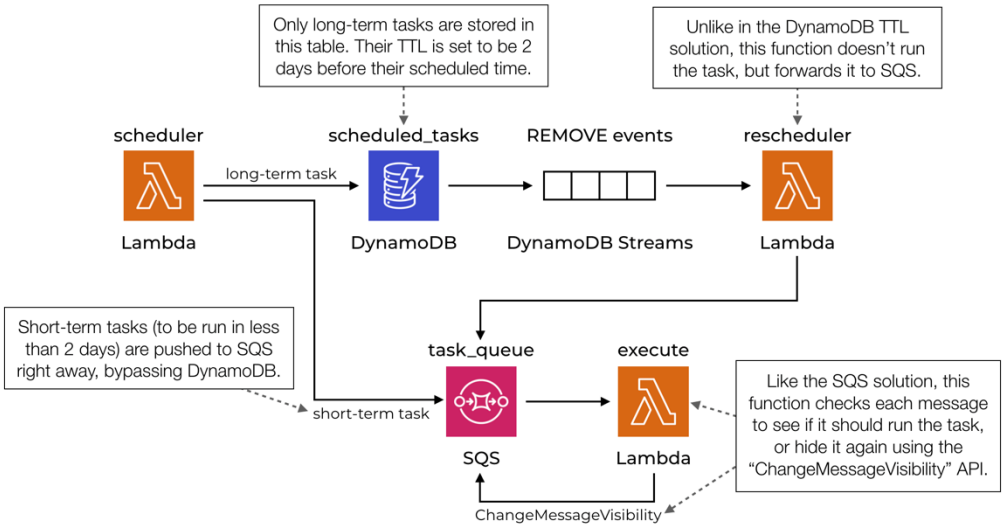


Figure 7.18 High level architecture of combining DynamoDB TTL with SQS.

7.7.1 Your scores

How would you score this solution? write down your scores in the empty spaces below.

	Score
Precision	
Scalability (number of open tasks)	

[16] https://amzn.to/2NRgARU

Scalability (hotspots)

Cost

7.7.2 My scores

The fundamental theorem of software engineering[17] says that

> "We can solve any problem by introducing an extra level of indirection."

And like the other alternative solutions we saw earlier in this chapter, this solution solves the problems with an existing solution by introducing an extra level of indirection. It does so by composes different services together in order to make up for the shortcomings of each.

So, here's my scores for this solution.

	Score
Precision	9
Scalability (number of open tasks)	8
Scalability (hotspots)	8
Cost	7

And here's how I arrived at these scores.

PRECISION

As all the executions go through SQS, this solution has the same level of precision as the SQS-only solution – 9.

SCALABILITY (NUMBER OF OPEN TASKS)

Storing long-term tasks in DynamoDB largely solves the SQS solution's problem with scaling the number of open tasks. However, it is still possible to run into the 120,000 inflight messages limit with just the short-term tasks. It's far less likely but is still a possibility that need to be considered. Hence why I marked this solution down here and only gave it an 8.

SCALABILITY (HOTSPOTS)

As all the executions go through SQS, this solution has the same score as the SQS-only solution – 8.

COST

This solution eliminates most of the `ChangeMessageVisibility` requests because all the long-term tasks are stored in DynamoDB. This cuts out a large chunk of the cost associated with the SQS solution. However, in return it adds additional costs for DynamoDB usage and Lambda invocations for the `reschedule` function.

Overall, I think the costs it takes away are greater than the new costs it adds. Hence why I gave it a 7 for Cost, improving on the original score of 5 for the SQS solution.

[17] https://en.wikipedia.org/wiki/Fundamental_theorem_of_software_engineering

7.7.3 Final thoughts

This is just one example of how different solutions, or aspects of them, can be combined to make a more effective solution. This combinatory effect is one of the things that makes cloud architectures so interesting and fascinating, but also, so complex and confusing at times.

There are so many different ways to achieve the same goal, and depending on what your application needs, there's usually no one-size-fits-all solution that offers the best result for all applications.

So far, we have only looked at the supply side of the equation and what each solution can offer. We haven't looked at the demand side yet, or what application needs what. After all, depending on the application, you might put a different weight behind each of the non-functional requirements. So let's try and match our solutions to the right application next.

7.8 Choosing the right solution for your application

These are my scores for the five solutions that we have considered (not including the proposed tweaks, table 7.3).

Table 7.3 My scores for all five solutions

	Cron job	DynamoDB TTL	Step Functions	SQS	SQS + DynamoDB TTL
Precision	6	1	10	9	9
Scalability (number of open tasks)	10	10	7	2	8
Scalability (hotspots)	2	6	4	8	8
Cost	7	10	2	5	7

Depending on the application, some of these requirements might be more important than others.

7.8.1 The applications

Let's consider three applications that might use the ad-hoc scheduling service:

1. Application 1 is a reminder app, let's call it **RemindMe**. In this app, users can create reminders for future events and the system would send SMS/push notifications to the users 10 mins before the event. While reminders are usually distributed evenly over time, there are hotspots around public holidays and major sporting events such as the Super Bowl. During these hotspots, the application might need to notify millions of users. Fortunately, because the reminders are sent 10 mins before the event, the system gives us some slack in terms of timing.

2. Application 2 is a multi-player mobile game called **TournamentsRUs**. Players compete in user-generated tournaments that are 15-30 mins long. As soon as the final whistle blows, the tournament ends and all participants would wait for a winner to be announced via an in-game popup. *TournamentsRUs* currently has 1.5 million daily active users (DAU) and hopes to double it in 12 months' time. At peak, the number of concurrent users is around 5% of its DAU, and tournaments typically consist of 10-15 players each.
3. Application 3 is a healthcare app that digitizes and manages your consent for sharing your medical data with care providers, let's call this app **iConsent**. Users can fill in digital forms that allow care providers to access their medical data. Each consent has an expiration date and the app needs to change its status to "expired" when the expiration date passes. *iConsent* current has millions of registered users and users have an average of 3 consents.

Each of these applications need to use the scheduling service to schedule ad-hoc tasks to run at specific times. But their use cases are drastically different. Some are dealing with tasks that are short-lived, others allow tasks to be scheduled for any point in time in the future. Some are prone to hotspots around real-world events, others can accumulate large number of open tasks because there is no limit to how far away tasks can be scheduled for.

To help us better understand which solution is the best for each application, we can apply a weight against each of the non-functional requirements for each application.

For example, *TournamentsRUs* cares a lot about precision because users will be waiting for their results at the end of a tournament. If the tasks to finalize tournaments are delayed then it can negatively impact the users' experience in the app.

7.8.2 Your weights

For each of the applications, write down a weight between 1 ("I don't care") and 10 ("This is critical") for each of the non-functional requirements. And remember, there are no right or wrong answers here! Use your best judgement based on the limited amount of information you know about each app.

	RemindMe	TournamentsRUs	iConsent
Precision			
Scalability (number of open tasks)			
Scalability (hotspots)			
Cost			

7.8.3 My weights

Ok, and here are my weightings.

	RemindMe	TournamentsRUs	iConsent
Precision	5	10	4
Scalability (number of open tasks)	10	6	10
Scalability (hotspots)	8	3	1

Cost	3	3	3

Are my scores quite similar to yours?

Let's take a moment and go through each application and talk about how I arrived at these weights.

RemindMe

I gave Precision a weight of 5 because reminders are sent 10 minutes before the event. This gives us a lot of slack. Even if the reminder is sent 5 minutes late, it's still OK.

I gave Scalability (number of open tasks) a weight of 10 because there are no upper bound on how far out the events can be scheduled. So at any moment in time, there can be millions of open reminders. Which makes scaling the number of open tasks an absolutely critical requirement for this application.

For Scalability (hotspots), I gave a weight of 8 because large hotspots would likely form around public holidays (for example, mother's day) and sporting events such as the Super Bowl.

Finally, for Cost, I gave a weight of 3. This perhaps reflects my general attitude towards the cost of serverless technologies. Their pay-per-use pricing allows my cost to grow linearly with my scale. So generally speaking, I don't want to optimize for cost unless the solution is going to burn down the house!

TournamentsRUs

For *TournamentsRUs*, I gave Precision a weight of 10 because when a tournament finishes, players would all be waiting for the winner announcement. If the scheduled task (to finalize the tournament and calculate the winner) is delayed for even a few seconds it would be a bad user experience.

I gave Scalability (number of open tasks) a weight of 6 because only a small percentage of its DAUs are online at once and the short duration of its tournaments. At 1.5M DAU, 5% concurrent users at peak and an average of 10-15 players in each tournament, these numbers translate to approximately 5000-7500 open tournaments during peak time.

For Scalability (hotspots) I gave it a lowly 3 because the tournaments are user-generated and have different lengths of time (between 15-30 minutes). So, it's unlikely for large hotspots to form under these conditions.

Just like with *RemindMe*, I gave Cost a weight of 3 – just don't burn my house down!

iConsent

Lastly, for iConsent I gave Precision a weight of 4. When a consent is expired, it should be shown in the UI with the correct status. However, because the user is probably not going to check the app every few minutes for updates, it's OK if the status is updated a few minutes (or maybe even an hour) late.

However, I gave a weight of 10 for Scalability (number of open tasks). This is because medical consents can be valid for a year or sometime even longer. And all active consents open tasks, so the system would have millions of open tasks at any moment.

For Scalability (hotspots) on the other hand, I gave it a weight of 1 because there is just no natural clustering that can lead to hotspots.

And finally, I gave cost a weight of 3 because that's just how I generally feel about cost for serverless applications.

7.8.4 Scoring the solutions for each application

So far, we have scored each solution based on its own merit. But it says nothing about how well suited a solution is to an application because as we have seen, applications have different requirements.

By combining a solution's scores with an application's weights, we can arrive at something of an indicative score of how well they are suited for each other. Let me show you how it can be done, and then you can do this exercise yourself.

If you recall, here are my scores for the cron job solution:

	Score (Cron job)
Precision	6
Scalability (number of open tasks)	10
Scalability (hotspots)	2
Cost	7

And for *RemindMe*, I gave the following weights:

	Weight (RemindMe)
Precision	5
Scalability (number of open tasks)	10
Scalability (hotspots)	8
Cost	3

Now, if we multiple the score with the weight for each non-functional requirement, we will arrive at the following:

	Weighted Score (Cron job x RemindMe)
Precision	6 x 5 = 30
Scalability (number of open tasks)	10 x 10 = 100
Scalability (hotspots)	2 x 8 = 16
Cost	7 x 3 = 21

Which adds up to a grand total of 30 + 100 + 16 + 21 = 167. On its own, this score means nothing very little. But if we repeat the exercise and score each of the solutions for *RemindMe*, then we can see how well they compare with each other. Which would help us pick the most appropriate solution for *RemindMe*, which might be different to the solution you would use for *TournamentsRUs* or *iConsent*.

So, with that in mind, use the whitespace below to calculate your weighted scores for each of the five solutions that we have discussed so far for *RemindMe*.

	Cron job	DynamoDB TTL	Step Functions	SQS	SQS + DynamoDB TTL

Precision	
Scalability (number of open tasks)	
Scalability (hotspots)	
Cost	
Total score	

And now repeat the exercise for *TournamentsRUs*.

	Cron job	DynamoDB TTL	Step Functions	SQS	SQS + DynamoDB TTL
Precision					
Scalability (number of open tasks)					
Scalability (hotspots)					
Cost					
Total score					

And finally, for *iConsent*.

	Cron job	DynamoDB TTL	Step Functions	SQS	SQS + DynamoDB TTL
Precision					
Scalability (number of open tasks)					
Scalability (hotspots)					
Cost					
Total score					

Did the scores align with your gut instinct for which solution is best for each application?

Did you find something unexpected in the process?

Did some solutions not fare as well as you thought it would?

If you find any uncomfortable outcomes as a result of these exercises then they have done their job. The purpose of defining requirements upfront and putting a weight against each requirement is to help us remain objective and combat cognitive biases.

7.9 Summary

Here are my weighted scores for each solution and application:

	RemindMe	TournamentsRUs	iConsent
Cron job	167	147	147
DynamoDB TTL	180	115	137

Step Functions	158	160	120
SQS	144	141	79
DynamoDB TTL with SQS	210	183	145

These scores give you a sense as to which solutions are best suited for each application. But they don't give you definitive answers and you shouldn't follow them blindly. For instance, there are often factors that aren't included in the scoring scheme but need to be taken into account nonetheless. Factors such as complexity, resource constraints and familiarity with the technologies are usually important for real-world projects.

As you brainstormed and evaluated the solutions that have been put in front of you in this chapter, I hope you picked up on the even more important lessons here: that all architecture decisions have inherit tradeoffs and not all application requirements are created equally. The fact that different applications care about the characteristics of its architecture to different degrees gives us a lot of room to make smart tradeoffs.

There are so many different AWS services to choose from, each offering a different set of characteristics and tradeoffs. When you use different services together, they can often create interesting synergies. All of these give us a wealth of options to mix and match different architectural approaches and engage in a creative problem solving process, and it's beautiful!

8

Architecting serverless parallel computing

This chapter covers

- **Principles of MapReduce**
- **Architecting a Serverless solution with Step Functions and EFS**

There's a secret about AWS Lambda that I like to tell people: It's a supercomputer that can perform faster than the largest EC2 instance. The trick is to think about Lambda in terms of parallel computation. If you can divide your problem into hundreds or thousands of smaller problems, and solve them in parallel, you will get to a result faster than if you try to solve the same problem by moving through it sequentially. *Parallel computing* is an important topic in computer science and is often talked about in the undergraduate computer science curriculum. Interestingly, Lambda, by its very nature, predisposes us to think and apply concepts from parallel computing. Services like Step Functions and DynamoDB make it easier to build parallel applications. In this chapter we'll illustrate how to build a Serverless video transcoder in Lambda that will outperform bigger and more-expensive EC2 servers. We'll look at the implementation, cost, performance, and discuss the pros and cons of the approach for decomposing and solving large problems using Lambda.

8.1 Introduction to MapReduce

MapReduce is a popular and well-known programming model often used to process large data sets. It was originally created at Google by developers who were themselves inspired by two well-known functional programming primitives (higher-order functions) – map and reduce. MapReduce works by splitting up a large dataset into many smaller subsets, performing an operation on each subset, and then combining or summing up to get the result. Imagine that you want to find out how many times a character's name is mentioned in "War and Peace."

You can sequentially look through every page, one by one, and count the occurrences. If you apply a MapReduce approach, however, you can do much quicker:

- First you **split** data into many independent subsets. In case of "War and Peace," it could be individual pages or paragraphs.
- The next step is to apply the **map** function to each subset. The map function, in this case, scans the page (or paragraph) and emits how many times the character's name is mentioned.
- There could be an optional **combine** step here to combine some of the data. It can help make the computation a little easier to perform in the next step.
- The **reduce** function performs a summary operation. It counts the number of times the map function has emitted the character's name and produces the overall result.

> **NOTE** It's important to understand that the power of MapReduce in the "War and Peace" example comes from the fact that the map step can run in parallel on thousands of pages or paragraphs. If this wasn't the case, then this program would be no different from a sequential count.

Figure 8.1 shows what a theoretical MapReduce architecture looks like.

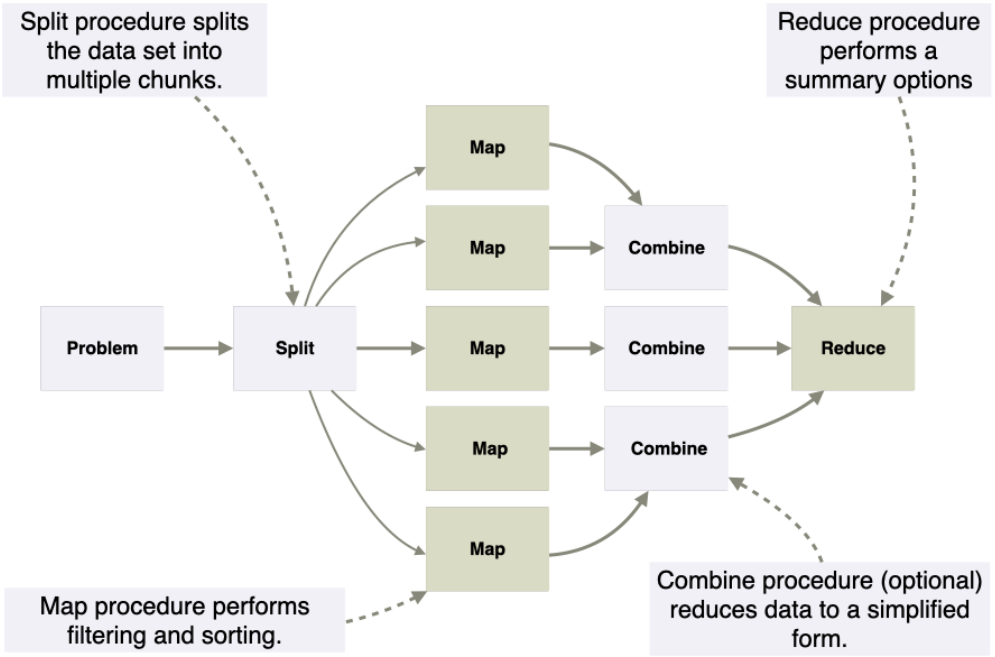


Figure 8.1 These are the steps a fictional MapReduce algorithm could take. We will build something like this soon.

As you have already guessed, real-world MapReduce application are often more complex. In a lot of cases, there are intermediary steps between map and reduce that combine or simplify data, and considerations such as locality of data become important in order to minimize overheads. Nevertheless, we can take the idea of splitting a problem into smaller chunks, processing them in parallel, and then combining and reducing them to achieve the outcome you need, and do it with Lambda.

8.1.1 How to transcode a video

In the first edition of this book, we built a YouTube clone using the AWS Elastic Transcoder. The Elastic Transcoder service takes uploaded videos and transcodes them from one format to another. This is useful because you can downscale videos or change their format. Ultimately, we built a video website with the Elastic Transcoder in the first edition, but Elastic Transcoder remained a black box. How does it work? What makes it tick? It's a managed service so we don't know.

> **TIP** If you were going to build a similar system now, we'd encourage you to use AWS Media Elementals rather than Elastic Transcoder. Media Elementals is more up to date, and more powerful.

So, our goal in this chapter is to implement our own video transcoder using Lambda. Our major requirements are the following:

1. Build a transcoder that takes a video file and produces a transcoded version in a different format or bitrate. We want complete control over the transcoding process.
2. Use only serverless services in AWS such as Lambda and S3. Obviously, we are not allowed to use an EC2 or a managed service to do transcoding for us.
3. Build a product that is robust and fast. It should be able to beat a large EC2 instance most of the time.
4. Learn about parallel computing and how to think about solving these problems.

The solution we are about to present works but it's not something we recommend running in a production environment. Unless your core business is video transcoding, you should outsource as much of the undifferentiated heavy lifting as possible in order to focus on your unique problem. In most cases, a managed service like AWS Media Elementals is better; you simply don't have to worry about keeping it running. So, take this as just an exercise and an opportunity to learn about MapReduce and parallel computation (and, you never know when you might face a big problem that requires the skill you may pick up here).

How would you do video transcoding in Lambda?

Take a moment and think about how you would build a video transcoder using AWS Lambda. All guesses are good, and we'd love to know (twitter.com/sbarski) how you would approach the problem. Can you build it yourself without reading the rest of the chapter?

8.1.2 The architecture

To transcode a file using Lambda we need to apply principles of MapReduce. But we are not implementing classical MapReduce here; instead we are taking inspiration from this algorithm to build our transcoder.

An interesting thing about Lambda (that we've mentioned before) is that it forces us to think parallel. You cannot process a 3GB video file in a Lambda function even if you wanted to treat a Lambda function like a traditional computer. You'd run out of memory and timeout quickly. The function would either stop after 15 minutes or exhaust all available RAM and crash. So, you have to think about how to decompose the problem into smaller problems that can be processed within Lambda's constraints. This leads you to a realization that your implementation needs to:

- divide the video file into a lot of tiny segments,
- transcode these segments in parallel,
- combine them together to produce a new video file

We need to parallelize as much as possible to get the most out of our serverless supercomputer. For example, if some segments are ready to be combined, while the others are still processing, we should combine the ones that are ready. Performance is the name of the game here. So, with that in mind, here's an outline for our serverless transcoding algorithm that, let's say, is designed to transcode a video from one bitrate to another:

1. A user uploads a video file to S3 that invokes a Lambda function.
2. That Lambda function analyzes the file and figures out how to cut up the source file to produce smaller video files (segments) for transcoding.
3. To make things go a little bit faster, we strip away the audio from video and save it to another file. Not having to worry about the audio, makes steps d-f faster to execute.
4. The next step performs the split process that creates small video segments for transcoding.
5. Now comes the map process that transcodes segments to the desired format or bitrate. The system can transcode a bunch of these segments in parallel.
6. The map process is followed by a combine step that begins to merge these small video files together.
7. The final step merges audio and video together and presents the file to the user. We have reduced our work to its final output.
8. As the kids say, the real final step is profit.

Here are the main AWS services and software that you will use to build the transcoder:

- **FFmpeg**: In the first edition of our book we briefly used ffmpeg – a cross-platform library/application created for recording, converting, and streaming audio and video. This is a powerhouse of an application that is used by everyone and anyone ranging from hobbyists to commercial tv channels. You are going to use ffmpeg in this chapter to do the transcoding, splitting, and merging of video files. You are also going to use a

utility called ffprobe to analyze the video file and figure out how to cut it up on keyframes.

DEFINITION A keyframe stores the complete image in the video stream whilst a regular frame stores an incremental change from the previous frame. Cutting on keyframes prevents us from losing information in the video.

- **AWS Lambda:** It goes without saying that you'll use Lambda for nearly everything. Lambda will run ffmpeg and execute most of the logic. You are going to write five functions to analyze the video, extract audio, split the original file, convert segments, merge video segments, and merge video and audio in the final step.
- **Step Functions**: To help us orchestrate this workflow you are going to rely on Step Functions. This service helps us to define how and in what order your execution steps will happen, makes the entire workflow robust, and provides additional visibility into the execution.
- **S3:** Simple Storage Service (S3) will be used for storing the initial and the final video. You could also use it to store the temporary video chunks, but you are going to use EFS for that. The reason why EFS is chosen is because it is easy to mount and access it as a file system from within Lambda. You can use S3, and we'll provide an alternative implementation that uses S3, but it is slightly harder to get right.
- **EFS:** The Elastic File System will be used by your Serverless transcoder to store the temporary files we generate. There will be a lot of small video files. Luckily, EFS can grow and shrink as needed.
- **DynamoDB:** Although Step Functions help to manage the overall workflow and execution of Lambda functions, we still need to maintain some state. We need to know whether certain video chunks have been created or can be merged. You are going to use DynamoDB to store this information. Everything that's stored is ephemeral and will be deleted using Dynamo's Time to Live (TTL) feature.

Figure 8.2 shows a high-level overview of the system you are about to build. We will jump into individual components in the next section.

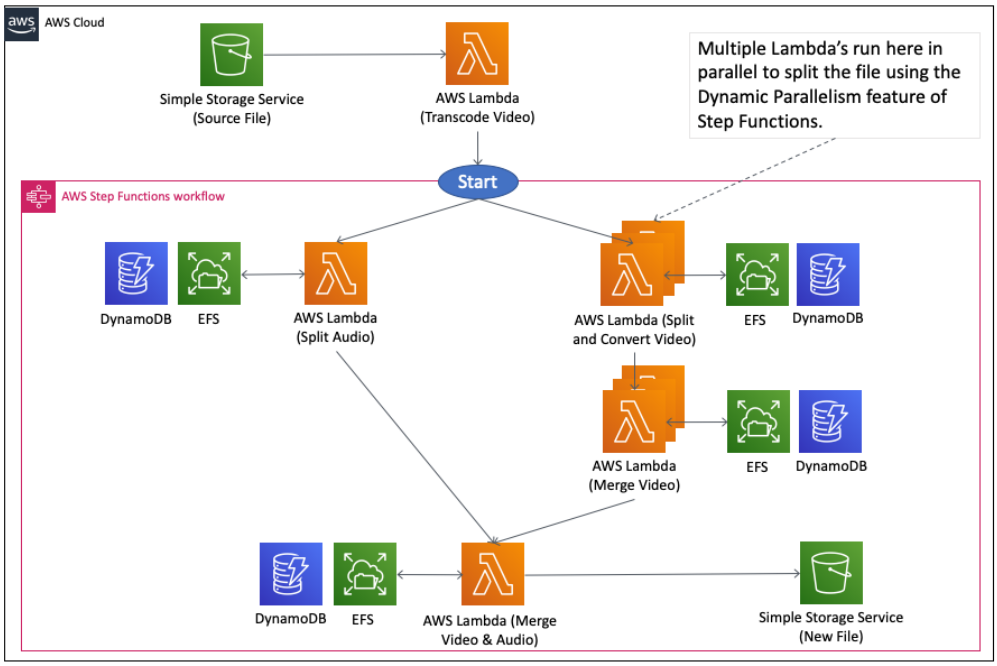


Figure 8.2 This is a simplified view of the serverless transcoder. There are a few more components to it but we've avoided included them to focus on the essential elements of the architecture.

8.2 Architecture deep dive

Let's explore the architecture in more detail now. There's nuance to the implementation and how things work. So, to avoid dealing with some pain later on, let's plan how we will design, build, and deploy the serverless transcoder. Before going any further, recall that the entire idea here is to split a giant video file into many small segments, transcode these segments in parallel, and then merge them together into a new file. Also, this is a good place to mention that you can grab the source code from: https://github.com/sbarski/serverless-video-transcoder. Currently, this implementation of the Serverless transcoder will convert files to 720p. But because we are using ffmpeg, you can do virtually anything. So, feel free to build the Serverless transcoder yourself or deploy our code and change it to your heart's content.

8.2.1 Maintaining state

DynamoDB is going to be used to maintain state across the entire operation of the serverless pipeline. It'll keep track of which videos have been created and which haven't. To simplify the pipeline and, in particular, to simplify the code that monitors which segments have been created or transcoded, we are going to use a trick. (Before going any further, think about how you would keep track of all small video segments that have been created, transcoded, and merged given that segments will be created and processed in parallel.)

The trick is to create n^2 smaller video segments. So, out of one large video file we should generate 2, or 4, or 8, or 256, or 512 or more segments. Just remember to keep the number of segments at n^2. Why is this? The idea here is that once we've created and transcoded our video segments, we can begin merging them in any order. Having n^2 segments easily allows us to identify which two neighbor segments can be merged. And, we can keep track of this information in the database. You are creating a basic binary tree that helps to make the logic around this algorithm a little easier to manage.

Let's imagine that you have 8 segments, here's how the process could take place:

- Segments 3 and 4 have been transcoded quicker than the rest and can be merged together (they are neighbors) into a new segment called 3-4.
- Then segment 7 is created but segment 8 is not yet available. The system waits for 8 to become ready before merging 7 and 8 together.
- Segment 8 is currently created and 7 and 8 can be merged together into a new segment 7-8.
- Then segments 1 and 2 finish transcoding and are merged into a segment 1-2.
- The good news is that a neighboring segment 3-4 is already available. Therefore 3-4 and 1-2 can be merged together into a new segment called 1-4.
- Segments 5 and 6 are transcoded and are merged into a segment 5-6.
- Segment 5-6 has a neighboring segment 7-8. These two segments are merged together to create segment 5-8.
- Finally, segments 1-4 and 5-8 can be merged together to create the final video which consists of segments 1 to 8.

Because we have n^2 segments, we can keep track of neighboring segments and merge them as soon as both neighbors (the left and the right) become available. Another interesting aspect is that segments themselves can figure out who their neighbors are for merging. A segment with an index that is cleanly divisible by 2 is always on the right, whereas a segment that is not cleanly divisible by 2 is the left neighbor. For example, a segment with an index of 6 is divisible by 2, therefore, we can figure out that it's on the right, and the neighbor it needs to merge with (when it becomes available) has an index of 5. Figure 8.3 illustrates how blocks can be merged together.

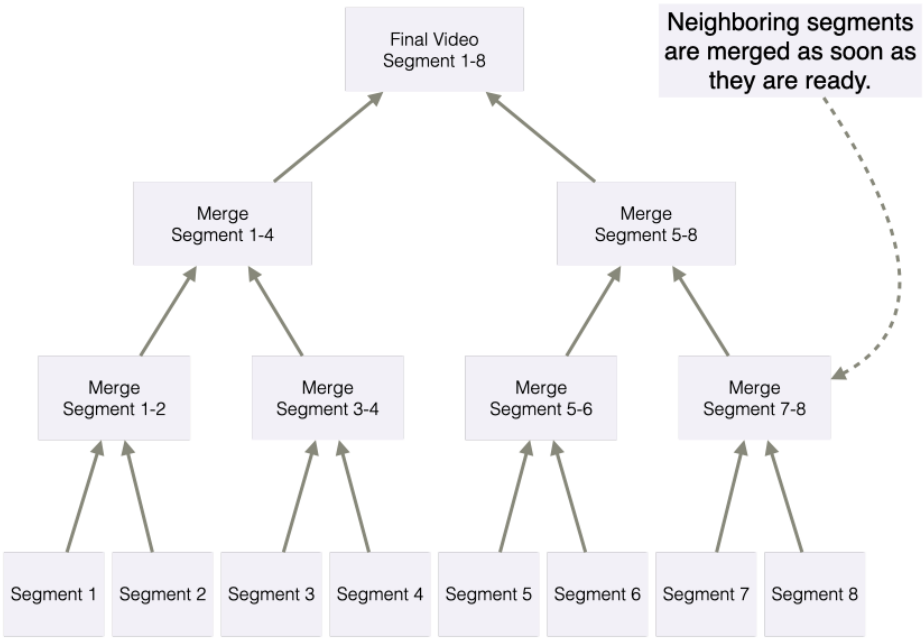


Figure 8.2 Segments are merged together into a new video. The beauty of our engine is that neighboring segments can be merged as soon as they are ready. There's no need to wait for other, non-related, segments to finish processing.

DynamoDB is an excellent tool for keeping track of which segments have been created and merged. In fact, you will pre-compute all possible segments and add them to DynamoDB. Then you will atomically increment counters in DynamoDB to have a record of when segments have been created and merged. That will allow the processing engine to figure out which blocks haven't been merged yet and which need to be done next. This is an important part of the algorithm so it's worth restating it again. Each record in DynamoDB represents two neighboring segments (for example, Segment 1 and Segment 2). The **Split and Convert** operation increment a **confirmation** counter each time a segment is created. So, when the confirmation counter is equal to 2, our system knows that the two neighboring segments exist and that they can be merged together. This information and logic are used in the **Split and Convert** function and in the **Merge Video** function. Your algorithm will continue merging segments and incrementing the **confirmation** counter in Dynamo until there's nothing left to merge.

There are more ways than one to do it

Our use of a binary tree is just one approach to solving this problem, keeping track of segments and ultimately merging them together. There are myriad other ways this can be done. How would you do it, if you had to come up with a different approach?

8.2.2 Transcode Video

Your Serverless transcoder will kick-off once you upload a file into an S3 bucket. An S3 event will invoke the **Transcode Video** Lambda and the process will begin. Figure 8.3 shows what this looks like.

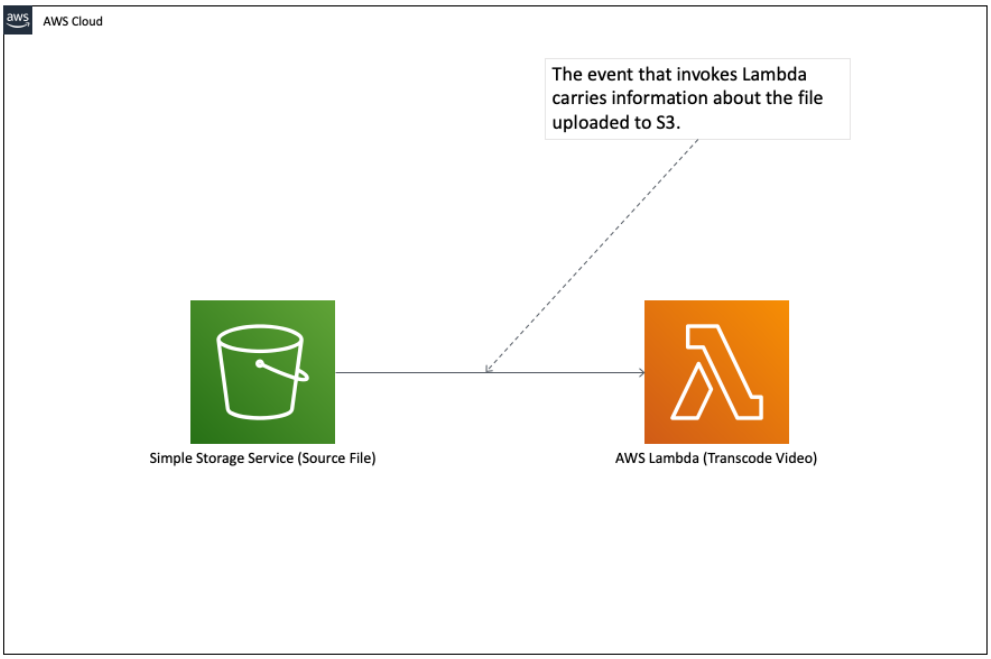


Figure 8.3 This is a very basic and common serverless architecture. Invoking code from S3 is the bread and butter of Lambda functions.

The transcode video function performs a number of steps:

1. It downloads the file from S3 to a local directory on EFS.
2. Analyzes the downloaded video file and extracts metadata from it. Video keyframes are provided in this metadata.
3. Creates the necessary directories in EFS for all future segments that are going to be created.
4. Works out how many segments need to be created based on the number of keyframes. However, remember that we are always creating n^2 segments. This means that we may have to create some "fake" segments in Dynamo. These will not

really do anything. They are considered as segments that have already been created. They don't need to be processed.

5. Creates the necessary records in DynamoDB including any "fake" records that are needed.
6. Runs a Step Functions workflow with two different inputs. The first parameter tells Step Functions to run a Lambda to extract and save the audio to EFS. The second parameter is an array of objects that specify the start and end times of all segments that need to be created. Step Functions takes this array and applies the Map procedure. It fans out and creates a Lambda function for each object in the array, thus causing the original file to be split up by many Lambda functions in parallel.

The transcode video function is an example of a "monolithic" or "fat" Lambda function because it does a lot of different things. It may not be a bad idea to split it up but that also comes with its own set of tradeoffs. In the end, whether you think this function should be kept together or not may depend on your personal taste and philosophy. We think that this function is a pragmatic solution for what we need to do but we wouldn't be aghast if you decided to split it up.

8.2.3 Step Functions

Step Functions play a central role in our system. This service orchestrates and runs the main workflow that splits the video file into segments, transcodes them, and then merges them. Step Functions also run a function that extracts the audio from the video file and saves it to EFS for safe-keeping. So, the main Lambda function that Step Functions invokes are:

- **Split Audio –** used to extract the audio from the video and save it as a separate file in EFS.
- **Split and Convert Video –** used to split the video file from a particular start point (for example, 5 minutes and 25 seconds) to an end point (such as 6 minutes and 25 seconds) and then encode the new segment to a different format or bitrate.
- **Merge Video –** used to merge segments together after they have been transcoded. Multiple Merge Video functions will run to merge segments until one final video file is produced.
- **Merge Video and Audio** – used to merge the newly created video file and the audio file to create the final output. This function uploads the new file back to S3.

> **PRO TIP** You don't have to extract the audio from the video and then transcode just the video file separately. We decided to do that because in our tests, our system ran a bit faster when the video was processed on its own and then recombined with the audio. However, your mileage may vary, so we recommend that you test video transcoding with and without extracting the audio first.

Step Functions is a workflow engine that is fairly customizable. It supports different states like Task (this invokes a Lambda function or passes a parameter to the API of another service) or Choice (this adds branching logic). The one important state you are going to use is Map. This state takes in an array and executes the same steps for each entry in that array.

Therefore, if you pass in an array with information on how to cut up a video into segments, Map will run enough Lambda functions to process all of those arguments in parallel. This is exactly what you are going to build. You will pass an array to a **Split and Convert** Lambda function using the Map type. Step Functions will create as many functions as necessary to cut the original video into segments.

But here comes the more interesting part. As soon as segments are created, Step Functions will begin calling the **Merge Video** function until all segments have been merged into a new video. You are going to add some logic to the Step Functions execution workflow to figure out if **Merge Video** needs to be called. Once all **Merge Video** tasks have been called and processed, Step Functions will take the result from **Extract Video** and from **Merge Video** and invoke the final **Merge Video and Audio** function. Figure 8.4 shows what this process looks like.

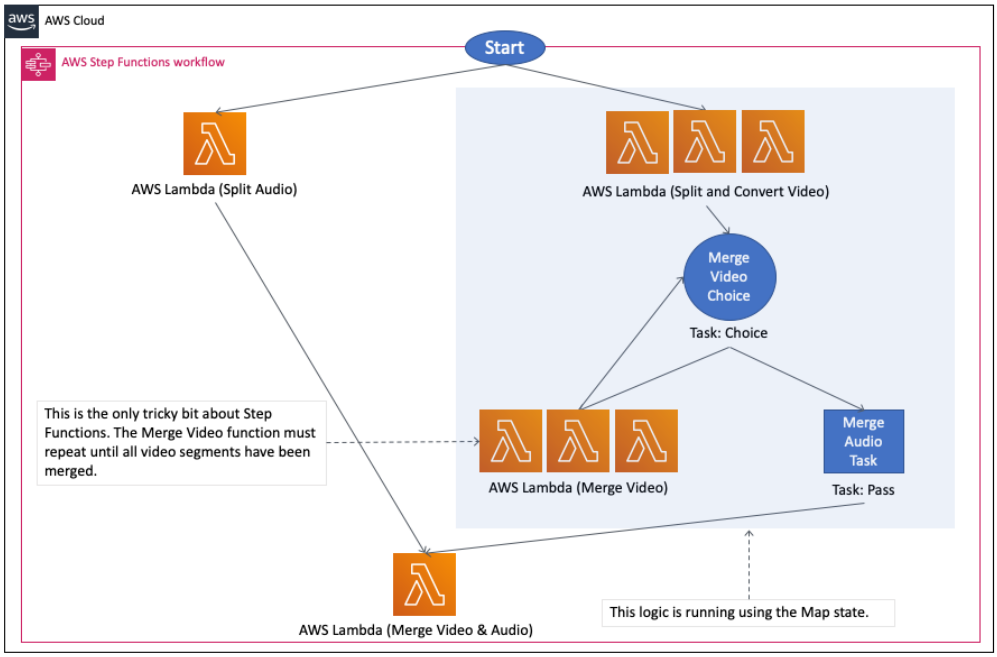


Figure 8.4 The Step Functions execution workflow does all the work in our transcoder. The video is split, converted, and merged again using two main functions and a bit of logic.

Now that you know what the Step Functions workflow does let's discuss each of the Lambda functions in more detail.

8.2.4 Extract Audio

The Extract Audio Lambda function is run by Step Functions to extract audio from the video file. As we've mentioned, this step is done to accelerate the overall workflow because from there on, the audio portion of the file isn't considered and only the video portion is transcoded to another bitrate. You don't have to do this, you can leave audio and video together, but in our case our testing showed that doing this improved the performance overall. The Extract Audio function executes the following steps:

- Extracts audio using ffmpeg and saves it to a folder in EFS.
- Updates the relevant DynamoDB record to record that this was done.
- Returns a "Success" message and additional parameters like the location of the audio file to the Step Functions orchestrator.

At a later stage, Step Functions invokes the **Merge Video Audio** function with the parameters that were returned by the **Extract Audio** and **Merge Video** functions.

8.2.5 Split and Convert Video

The **Split and Convert Video** function splits the original video file into a segment and converts that segment to a new bitrate or encoding. The original video file doesn't get changed in this process; instead, the function merely extracts a segment between a start time and an end time specified in the parameters that are passed to it. These parameters are worked out by the **Transcode Video** function. Many hundreds of **Split and Convert Video** functions can run in parallel. Here are the main actions that it performs:

- Using FFmpeg, the function creates a new video file from the original one.
- It increments a confirmation counter in the appropriate DynamoDB record to specify that the segment exists.
- If the confirmation counter is equal to 2 it then returns to the Step Functions workflow with a "Merge" message. Otherwise, it returns with a "Success" message, which stops the execution of that particular Step Functions parallel execution.

You may recall from the previous section that in DynamoDB, each record represents two neighboring segments. So, when a record counter is incremented to 2, the function knows that the two neighboring segments exist. The function returns a "Merge" message to Step Functions, and Step Functions knows that it can begin calling the **Merge Video** for these segments.

8.2.6 Merge Video

The **Merge Video** function is called whenever two neighboring segments are ready to be merged into a new single segment. The merge operation happens using ffmpeg and the new segment is saved to EFS. Here's what happens in a little more detail:

- The function is invoked with a number of parameters passed to it by Step Functions. These parameters include the left and the right segment.
- Using ffmpeg, left and right segments are merged to create a new segment. This new segment is saved to EFS.

- DynamoDB confirmation is incremented. If there are two confirmations, then the function returns to the workflow with a "Merge" message. However, if there are two confirmations and the last two remaining segments have been merged, the function returns with a "MergeAudio" message to the workflow.

As you can see, the **Merge Video** function creates a bit of a loop. It continues to merge segments, returns the Merge message, and causes Step Functions to invoke itself again. This happens until the last two segments are merged, when the return type is changed to "MergeAudio." This is when Step Functions know that it's time to combine audio and video and invoke the **Merge Video and Audio** function.

8.2.7 Merge Video and Audio

The final function is **Merge Video and Audio**. It takes input from **Extract Audio** and **Merge Video** and merges the audio and the new video files together using ffmpeg. At the end, it uploads the new file to another S3 bucket using the streaming capability of the S3 SDK.

Now you understand the main principles of the Serverless transcoder. All of the source code is available on GitHub at https://github.com/sbarski/serverless-video-transcoder. You can deploy it by installing the Serverless Framework on your computer and then running *sls deploy*. However, we encourage you to try to build your own serverless transcoder first. It is fun to do and will ultimately be much more educational. Reference our architecture and build it yourself. Otherwise grab the code and give it a go. We are looking for improvements to it too so if you have ideas on how to make it better, we'd love to hear from you (twitter.com/sbarski) and make those improvements.

8.3 Alternative Architecture

You can build this Serverless Transcoder without using EFS or Step Functions for that matter. In fact, our first iteration used only S3 and SNS to perform fan-out. We wanted to present to you an alternative architecture that shows that you don't necessarily have to use Step Functions or EFS if you don't want to. This section demonstrates that you can use SNS and S3 instead and achieve the same outcome. It's nice that AWS provides so many building blocks that we can build our desired architecture in different ways.

> **PRO TIP** One reason for adopting a different architecture could be because you don't want to pay for Step Functions and EFS. That is a reasonable concern. Using S3 is likely going to be much cheaper than using EFS and will probably perform just as well. Once you get the code working, using S3 is straightforward and we don't really have a reason to not recommend it. Whether you should use SNS instead of Step Functions is a tougher proposition. SNS will be cheaper but you will lose a lot of the robustness and observability that you get with Step Functions. Perhaps the best solution is to use Step Functions with S3? We'll leave it to you as an exercise to achieve.

This implementation will closely resemble the implementation you created in the previous section except, as we mentioned, we'll replace Step Functions with SNS, and EFS with S3. Figure 8.5 shows what this architecture looks like.

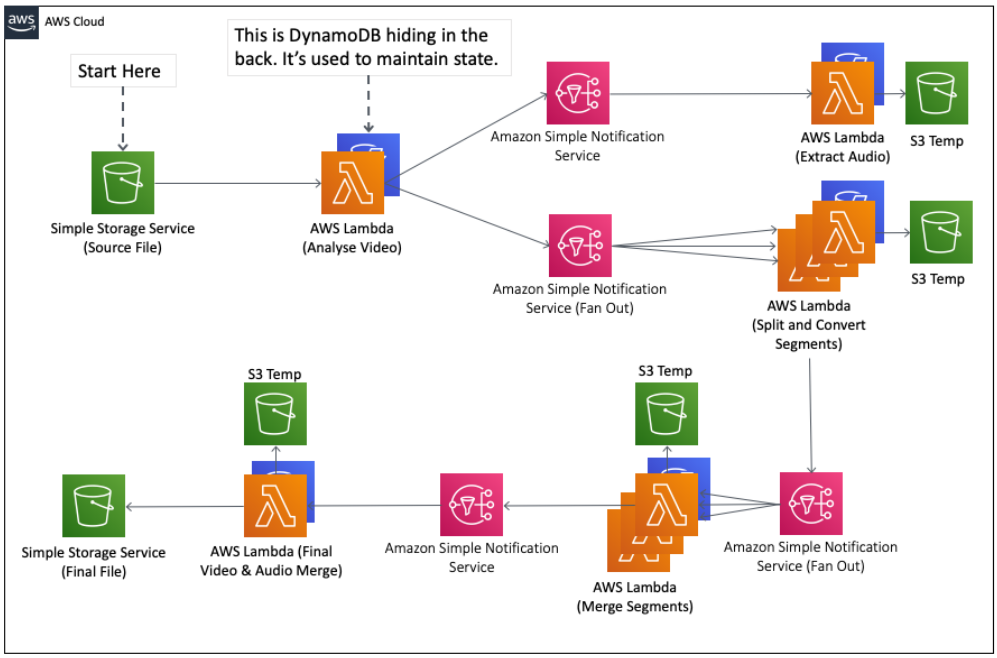


Figure 8.5 The SNS and S3 architecture for the Serverless Transcoder.

This architecture works well but there are some improvements that can be made to it. For one, the implementation should be improved in case of errors, such as the split or merge operation failing. Luckily there is the Dead Letter Queue (DLQ) feature of Lambda that allows you to save, review, and even replay failed invocations. If you want a challenge, we invite you to implement DLQ for this architecture to make it more resilient to errors. The second issue is observability and knowing what's happening with the system. Step Functions provides some level of visibility, but things get a little bit harder with SNS. One tool you can use is X-Ray. This AWS service can help you understand the interactions of different services within your system. It goes without saying that CloudWatch is essential too.

8.4 Summary

- MapReduce can work really well with a Serverless approach. Lambda invites you think about parallelization from the start, so take advantage of that.
- You can solve a lot of problems in Lambda and process vast amounts of data by splitting it up into smaller chunks and parallelizing the operations.
- Step Functions is an excellent service for defining workflows. It allows you to fan-out and fan-in operations.
- EFS for Lambda is an endless local disk. It grows as much as you need. You can run applications with EFS and Lambda that you couldn't have run before. Having said that,

S3 is still likely to be cheaper so make sure to do your calculations and analysis before choosing EFS.

- You can solve problems in different ways.
    - You don't have to use Step Functions because you could use SNS (although Step Functions adds an additional level of robustness and visibility).
    - You don't need to use EFS because you can use S3.
- When coming up with an architecture for your system, explore the available options because there will be different alternatives with different tradeoffs.

12

Blackbelt Lambda

This chapter covers

- Monitoring latency, request per second, and concurrency for serverless applications
- Techniques for optimizing latency

Performance (how fast your application responds) and availability (whether or not your application provides a valid response) are critical aspects of your end user experience. When using serverless architectures, your performance also has a direct impact on your costs; for example, AWS Lambda bills you for the duration your function runs, weighted by the memory you assign to it. Serverless architectures eliminate many of the common surface areas for performance optimizations, like scaling available servers or tweaking server configurations, which can make it challenging for new users to understand how to go about making these optimizations. This chapter introduces you to key tools and approaches available to you to improve performance across the various services that make up your serverless application. You'll use relevant examples to demonstrate how these techniques work.

12.1 Where to optimize?

Before we delve into how we optimize serverless architectures, let's quickly recap how to think about them. Serverless architectures have multiple conceptual layers, as illustrated in figure 12.1. **Endpoints** are responsible for secure interactions with your end users and devices, and for the ingress of requests or events to your application from the end user. Examples of endpoints you can use in your AWS Serverless architecture include API Gateway (covered in chapter XX), AWS IoT, Amazon Alexa (if you were building an Alexa skill), or even just the AWS SDK. The **compute** layer of your workload manages requests from external systems (received through the endpoints), while controlling access and ensuring requests are appropriately authorized. It contains the *runtime environment* that deploys and runs your business logic embodied as Lambda functions (we'll delve into this shortly). The **data layer** of

your workload manages persistent storage from within a system. It provides a secure mechanism to store states that your business logic will need. It also provides a mechanism to trigger events in response to data changes, which in turn can feed into other parts of your business logic. As you can imagine, this is very broad surface area to discussion optimizations across, so we are going to focus on the following:

- **functions**
- **invocations** of these functions (either via requests from endpoints or events from backend systems
- **interactions** the functions have with downstream resources. These points are highlighted in the following figure.

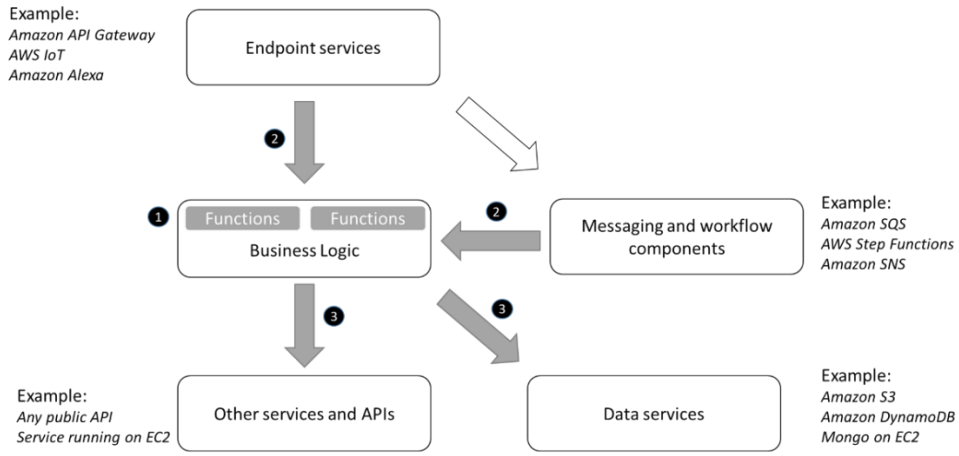


Figure 12.1 Conceptual architecture of a serverless application

Now that you have a conceptual understanding of the various points of optimization, let's next look at the tools available to do so.

12.2 Before we get started

To effectively optimize applications, there are certain tools and concepts we must be familiar with. In this section, we will recap what happens when a Lambda function executes and how it impacts latency, how to observe the latency and contributors to it, and how to generate load to a function to get enough sample data.

12.2.1 How a Lambda function handles requests

To understand how to optimize functions, we need to have a shared understanding of how Lambda goes about executing your function. Let's use an example to illustrate what happens when a function is deployed. We'll use the image-resizer-service application from the Serverless Application Repository

(https://serverlessrepo.aws.amazon.com/applications/arn:aws:serverlessrepo:us-east-1:526515951862:applications~image-resizer-service) for reference. This serverless application deploys a Lambda function (written in Node.js) and API Gateway to your AWS account in the US-east-1 region, that reads images from a S3 bucket (whose name defined at deployment) and serves them through API Gateway. The function uses the ImageMagick library to process the image.

> **REMEMBER** You need to specify a new bucket name for the application to use. Use "image-resizer-service-demo" for this example.

Once deployed, click the "test app" button on the page and it will take you to the application list view (1) on the Lambda console, where you'll see the newly deployed application (2).

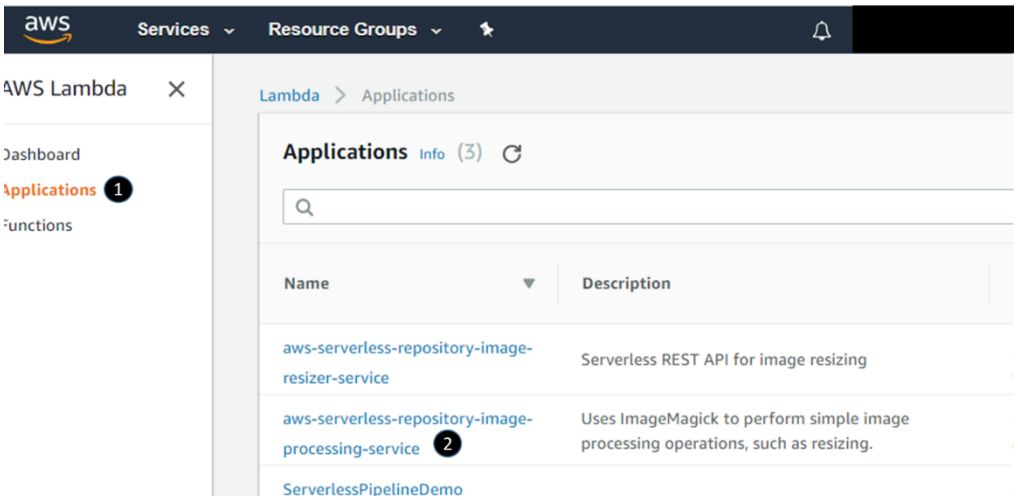


Figure 12.2 Application view

To test the application, you need to navigate to the main function. Click the application, and on the detailed view, click the "image resize" function (1) to access the function.

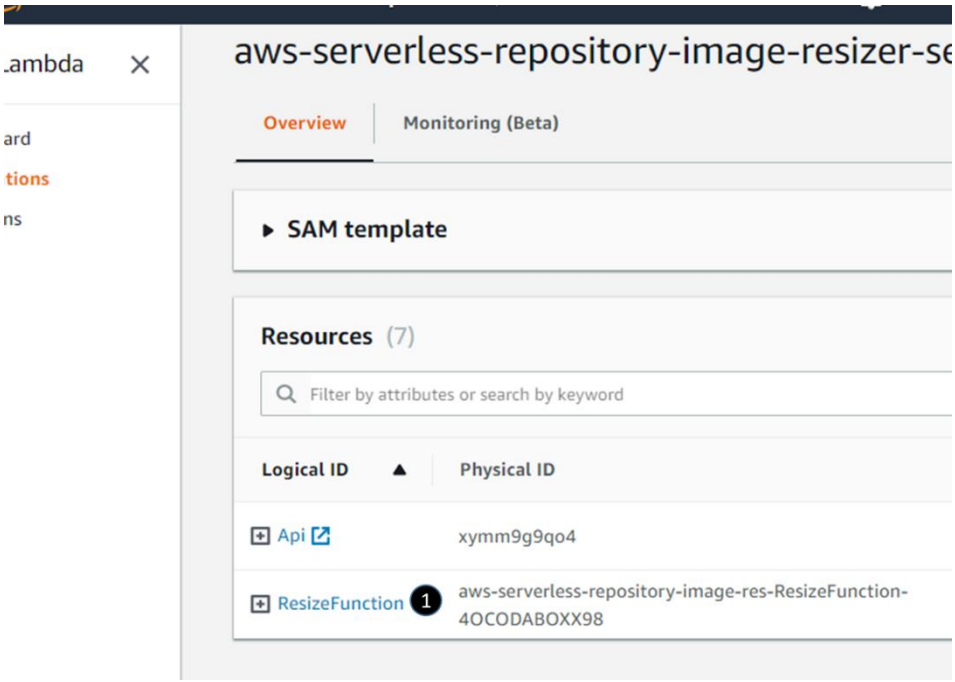


Figure 12.3 Application detail

Once you select the function, you are taken to the function detail page. Here, you can test the function by using the "test" functionality (1), but you need to configure a sample event to supply to the function first (2).

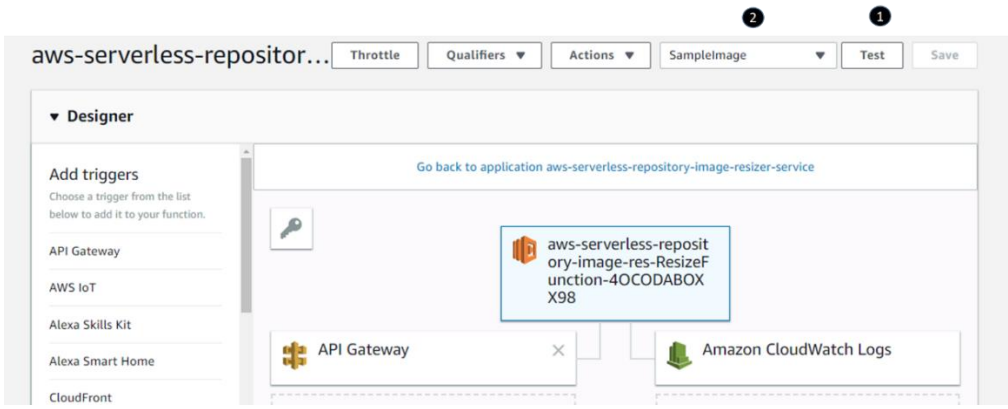


Figure 12.4 Function detail

You can use the test event in listing 12.1 to test the function. However, first upload a file https://commons.wikimedia.org/wiki/File:Yellow_Happy.jpg to the image-resizer-service-demo bucket (if you chose to upload a different image, be sure to change the object name in this listing, named in the "key" field). You need to do this so that the function doesn't error out looking for an object that doesn't exist!

Listing 12.1 Sample event

```
 {
"Records": [
  {
   "eventVersion": "2.0",
   "eventTime": "1970-01-01T00:00:00.000Z",
   "requestParameters": {
    "sourceIPAddress": "127.0.0.1"
   },
   "s3": {
    "configurationId": "testConfigRule",
    "object": {
     "eTag": "0123456789abcdef0123456789abcdef",
     "sequencer": "0A1B2C3D4E5F678901",
     "key": " Yellow_Happy.jpg",
     "size": 1024
    },
    "bucket": {
     "arn": "arn:aws:s3::: image-resizer-service-demo ",
     "name": " image-resizer-service-demo ",
     "ownerIdentity": {
      "principalId": "EXAMPLE"
     }
    },
    "s3SchemaVersion": "1.0"
   },
   "responseElements": {
    "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/mnopqrstuvwxyzABCDEFGH",
    "x-amz-request-id": "EXAMPLE123456789"
   },
   "awsRegion": "us-east-1",
   "eventName": "ObjectCreated:Put",
   "userIdentity": {
    "principalId": "EXAMPLE"
   },
   "eventSource": "aws:s3"
  }
 ]
}
```

Invoke the function a few times to evaluate the behavior; you will use this function to discuss the various optimizations that follow.

When you invoke this function, there are different layers in play—the Lambda compute *substrate*, the *execution environment*, and the function code (figure 12.5). The substrate is invisible to you, the execution environment is instantiated on demand for scale events (like a burst of requests), and the function code is instantiated for every request.

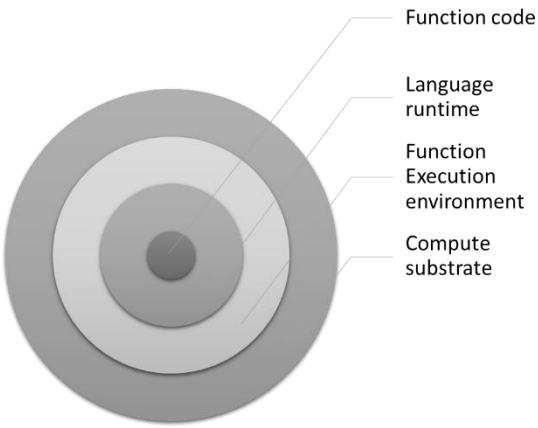


Figure 12.5 Layers involved in executing a function

When the first request or event arrives for your function, the AWS Lambda service performs a series of steps (figure 12.6), as follows:

1. Downloads your Lambda function Node code onto the part of the compute substrate where your code will run
2. Instantiate a new execution environment (sized based on your function allocation) with a Node runtime
3. Instantiates your non-function dependencies (in this case, ImageMagick)
4. Runs the parts of your function written outside the handler (we don't have any in this example)

Once the environment exists, Lambda runs the code inside your function handler.

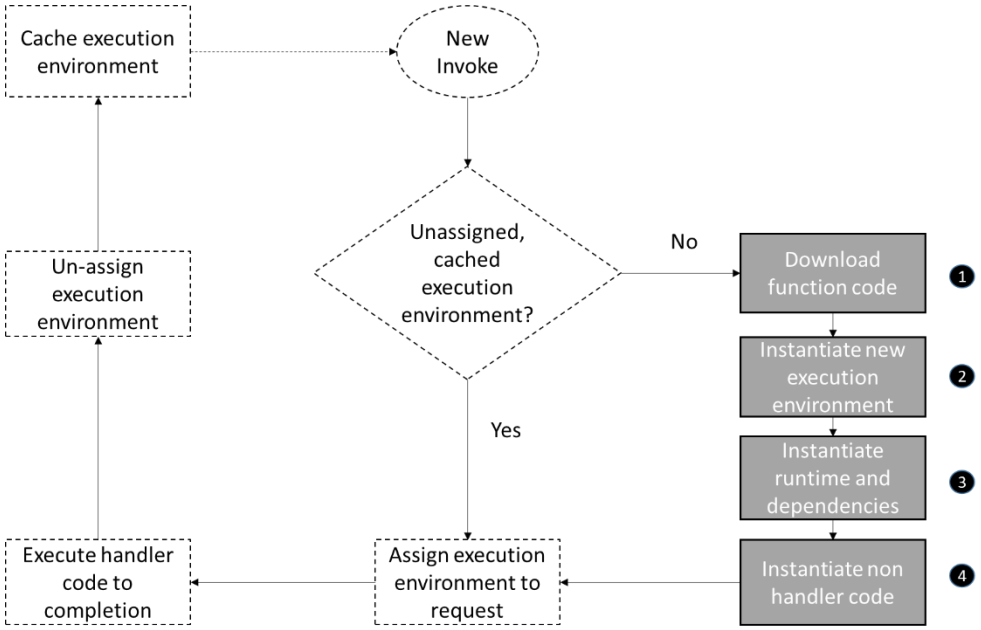


Figure 12.6 Request lifecycle

In our example (the "Resize Function), When your function handler runs, it processes the image and returns the image metadata. Lambda considers the function as done processing the request when the handler logic (and any threads spawned from within the function handler) finishes executing. When the request is complete, however, AWS Lambda does not discard the execution environment (with the runtime and code initialized). Instead, it caches the execution environment, where all processes inside the execution environment are paused. AWS does not publish any official guidance on how long the environment is retained in this state, but various published experiments show this ranging from 5-20 minutes.[1]

When a subsequent request arrives during this time and a cached execution environment is available, AWS Lambda will reuse that execution environment to service the request. On the other hand, if a cached execution environment is not available, AWS Lambda will repeat all the steps to serve the request. This has significant implications to both the performance of your function and how you write your function; we'll discuss this further in this chapter. One important behavior to remember is that AWS Lambda always runs only one request per execution environment. This means that if all execution environments are processing requests and a new one comes in, AWS Lambda will instantiate a new execution environment.

[1] https://www.usenix.org/conference/atc18/presentation/wang-liang

12.2.2 Latency – cold vs. warm

The latency incurred due to steps 1 through 4 in this example is referred to commonly as the ***cold start*** penalty. We refer to the request latency for a request incurring a cold start as *cold* latency, and we refer to the actual function execution latency as the ***warm*** latency. As a reminder, you incur the cold start penalty only in two situations. First, you'll see cold starts if your function has never been invoked before or is being invoked after an extended period (such that all cached execution environments are removed).

Second, you'll see cold starts if there is an increase in the incoming request rate such that AWS Lambda needs to spawn new execution environments because all available ones are servicing requests. For most production scenarios, cold starts impact less than 0.5% of requests, but cold starts disproportionally impact functions that are invoked infrequently and functions having a burst of traffic (specifically for the requests that first lead to the increased traffic). Requests that experience cold starts may also have experience timeouts, because The AWS Lambda timeout setting is applied to the total request latency.

12.2.3 Load generation on your function and application

As you go about optimizing your application, you want to do so at a load representative of real-life usage. As you can see, your latency characteristics may vary based on load as well. *Serverless-artillery* is a Nordstrom open-source project. It builds on artillery.io and serverless.com by using the horizontal scalability and pay-as-you-go nature of AWS Lambda to instantly and inexpensively throw arbitrary load at your services and report results to an InfluxDB time-series database (other reporting plugins are available). This capability gives you performance and load testing on every commit early in your CI/CD pipeline, so performance bugs can be caught and fixed immediately. https://github.com/Nordstrom/serverless-artillery-workshop presents a detailed walk through on using and setting up the tool.

12.2.4 Tracking performance and availability

You can't optimize what you can't measure. Before you go about figuring out how to reduce the latency and improve the availability of your serverless application, you must have a consistent approach to monitor this information. AWS offers a variety of both native and third-party tools for this task. To see what's available, pick any of your functions on the AWS Lambda console, click the function in the function list in the AWS Lambda console, and navigate to the Monitoring Tab (figure 12.7). You'll see three tools available to you out of the box – CloudWatch metrics (1) on the selected page, CloudWatch logs (2), and AWS X-Ray (3).

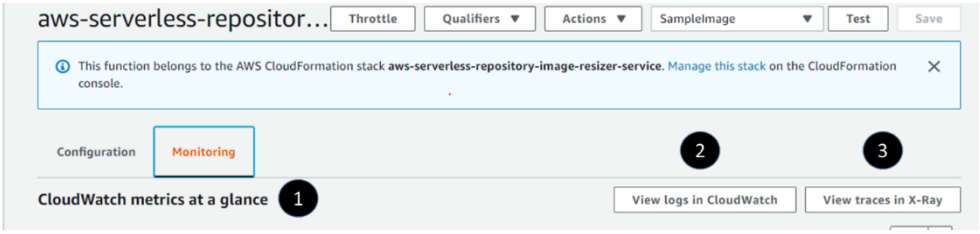


Figure 12.7 Monitoring tab for AWS Lambda functions

In this chapter, we'll be using CloudWatch metrics and X-Ray as the two primary tools to observe the latency characteristics of the application.

CloudWatch metrics

Each Serverless service (like AWS Lambda and API Gateway) emits standard metrics that help understand the performance and availability characteristics. For Lambda, AWS offers the following metrics among others:

1. *Invocations* – Total number of requests received by the given function. This is inclusive of all requests, independent of whether they were processed successfully, throttled, or resulted in an error. This also includes any requests that were retried due to Lambda's built in retry policy (more on this later).
2. *Duration* – Measures the elapsed wall clock time from when the function code starts executing because of an invocation to when it stops executing. This is a reasonable proxy for what your function will be billed, though not exact because AWS Lambda rounds your billed duration to the nearest 100ms.
3. *Errors:* Measures the number of invocations that failed due to errors in the function. Note that this does NOT measure errors due to problems in the AWS Lambda service, or due to throttling.
4. *Throttles:* Measures the number of invocations that did not result in your function code executing, because your function hit either it's concurrency limit, or caused the account to hit its concurrency limit.

AWS X-Ray

AWS X-Ray is a service that allows you to detect, analyze, and optimize performance issues with your AWS Lambda functions, as well as trace requests across multiple services within your serverless architecture. X-Ray generates traces for a sample of requests that each function receives, where a trace consists of segments for each service that the request traverses. A segment may further contain subsegments that detail what particular aspect of the service added to the latency of the request. To turn on X-Ray, you must click the "enable active tracing" checkbox under "Debugging and error handling" on the function detail page. As an example, figure 12.8 shows the trace for a simple sample application. You can see the total time spent in Lambda (1), the time your function took to execute (2), as well as the time spent

in a "cold start" (3). X-Ray can be a useful tool to determine where the bottlenecks in your function execution are to optimize, including whether the top contributor is a cold start.

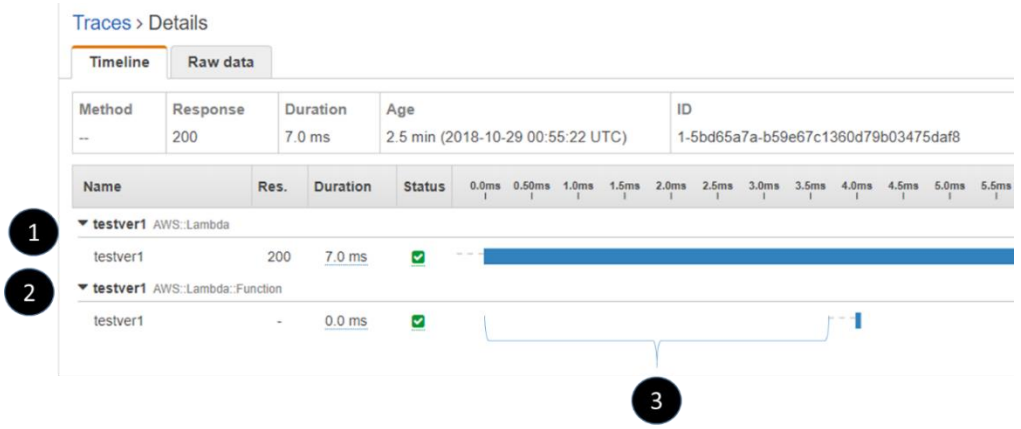


Figure 12.8 X-Ray trace for sample application

THIRD-PARTY TOOLS

There's a growing ecosystem of non-AWS tools that can also be used for performance and availability monitoring, including well established companies like NewRelic (https://newrelic.com/) and serverless-first companies like IOPipe (https://www.iopipe.com/) and Epsagon (https://epsagon.com/). We won't dive deep into these tools in this chapter but encourage you to explore all options and choose what works best for you.

A note on CloudWatch logs

As discussed in earlier chapters, CloudWatch logs capture any log activity specified within a Lambda function. CloudWatch logs can also be used in two additional ways:

- *As a data source for custom metrics*: For example, you can emit data points for the time spent within a specific method of your Lambda function and visualize and alarm on that information as a custom metric in CloudWatch metrics.
- *As a bridge to surface data to third-party tools:* CloudWatch logs makes it easy to send data to third-party tools like NewRelic or IOPipe, which in turn can provide additional visualization and tracing. Because Lambda does not support the inclusion of third-party agents directly in the code today, CloudWatch logs is the easiest way to surface operational information to other services.

12.3 Optimizing latency

You now have an understanding of what contributes to your application latency, how to generate load to your application to observe the latency, and what tools to observe the latency. In this section, we discuss how to improve it. You best return on effort at optimizing latency is

within individual functions. As the core glue and logic component of your application, any changes made to the function can have direct and immediate impacts to the latency that your customers experience, and to your overall application costs. For example, reducing function execution time by 10% reduces the cost of the function by 10%, which can be significant at high scale. The decision on what percentile and number to optimize for is your choice, depending on your customers. For example, if you are building a website, you want your response time to be less than 2s at the 99th percentile; if you are running a backend API, you may be able to tolerate 10s of seconds of response time at the 99th percentile.

12.3.1 Minimize deployment artifact size

The size of your deployment package directly impacts your cold start penalty in two ways. As a reminder, one of the steps that AWS Lambda undertakes on a "cold" invoke is downloading your code (step 1).

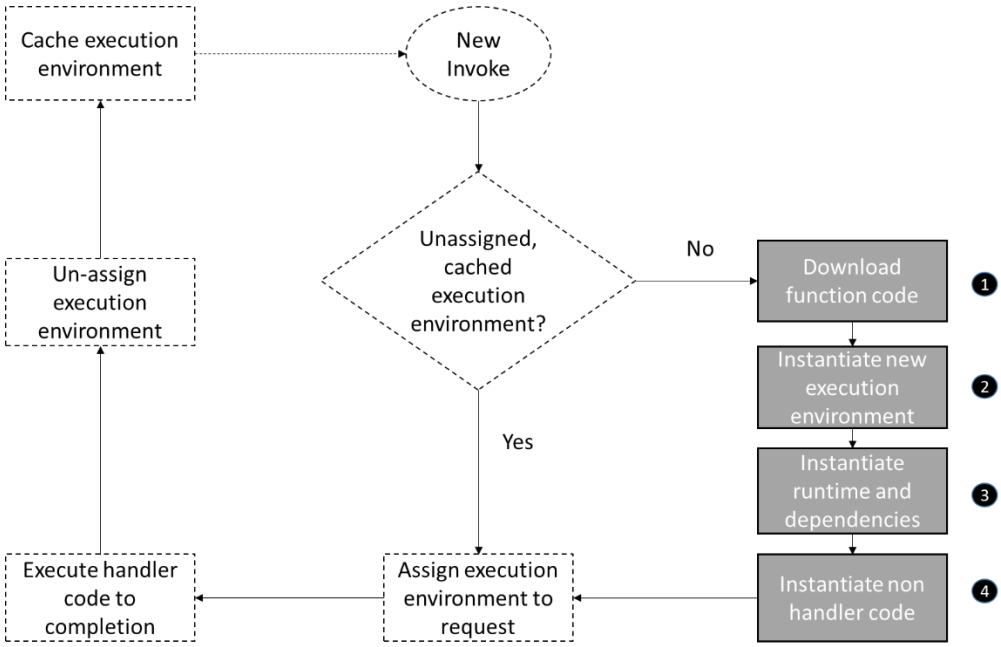


Figure 12.9 Request lifecycle

The larger your function, the longer this step takes – it's that simple! AWS Lambda enforces a limit of 250MB for your functions-deployment package, so there's a natural "worst case" impact for your deployment package. Second, for functions written in compiled languages like Java and C#, larger deployment packages with many dependencies take longer to instantiate when there are many classes to load into the CLASSPATH. As an example, a simple "hello world" on

Java loads only 429 classes in the JVM in about 0.1 seconds, while doing the same "hello world" using Clojure loads 1988 classes – three times as much and taking about 1s.

A best practice to follow is to audit any function dependencies. Are there any heavyweight library dependencies be removed or lightweight versions that can be used? Especially look for libraries that act as HTTP servers or agents; they have no use inside Lambda functions, because Lambda acts as the server for you. For example, instead of using the default Java spring library, you can use the streamlined https://github.com/awslabs/aws-serverless-java-container library which is approximately 30% faster in experiments. In our example, instead of packaging the entire AWS SDK, you could include only the SDK required for accessing S3. You can audit your dependencies for Node using tools like https://npm.anvaka.com/, or for Python using https://pypi.org/project/modulegraph/, or for Java using the Maven dependency tree.

Languages also offer specific tools to reduce deployment package sizes. For example, you can use minify for Node (https://www.npmjs.com/package/node-minify) to reduce the overall size of your Node function package. You can also use ProGuard (https://www.guardsquare.com/en/products/proguard) to reduce the size of your Java deployment package (JAR files).

12.3.2 Allocate sufficient resource to your execution environment

You code requires compute resources (CPU, memory) to run. AWS Lambda provides a single dial to set the resources required by your function – the *memory* setting. You can change this setting by navigating to the function detail page on the AWS console, go to the "basic settings" section of the Lambda console, and experiment with the slider for memory allocation. You can also set the same value via the API and CLI, but for simplicity you'll use the console in this example.

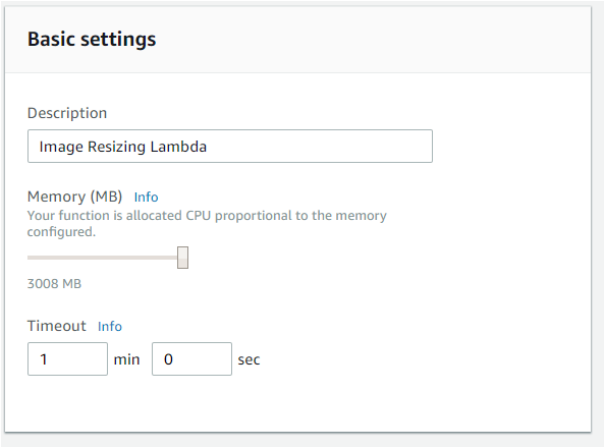


Figure 12.10 Function Memory setting

AWS Lambda allocates CPU power proportional to the memory by using the same ratio as a general-purpose Amazon EC2 instance type such as an M3 type. For example, if you allocate 256 MB memory, your Lambda function will receive twice the CPU share than if you allocate only 128 MB. You can update the configuration and request additional memory in 64-MB increments from 128 MB to 3008 MB. This change is not free: AWS Lambda pricing weights the billed duration for your function by its memory setting. So, 1 second of function execution time at 1024 MB costs the same as 8 seconds of execution at 128 MB.

Let's experiment with the memory setting on the image-resizer-service function you created so you can see the impact (if you haven't, see the "a Lambda function handles requests" section earlier in this chapter). Set the memory to 128MB, 256MB, 512MB, and 1024MB and run a few test invokes using the console (recommend at least 10). Now note the average execution time for those invocations from CloudWatch metrics. You should see results similar to that in the following table. The estimated costs are based on AWS Lambda public pricing for 1000 requests to the function.

Table 12.1 Estimated costs for 1000 requests

Memory	Duration	Estimated cost for 1000 requests
128 MB	11.722965sec	$0.024628
256 MB	6.678945sec	$0.028035
512 MB	3.194954sec	$0.026830
1024 MB	1.465984sec	$0.024638

We see that increasing the memory in this case kep the cost relatively flat, while increasing the performance ~10x. You'll typically see this kind of gains for for CPU-bound functions like image processing – more resources can help the function run faster without changing the costs. For I/O-bound operations (such as those waiting for a downstream service to respond), you'll see no benefit in increasing the resource allocation. For lightweight runtimes like Node and Go, you may be able to reduce the setting to the lowest (128MB); for runtimes like Java and C#, going lower than 256B can have detrimental effects to how the runtime loads your function code.

Finding the right resource allocation for your function will require some experimentation. The easiest path is to start with a high setting and reduce it till you see a change in performance characteristics. If you are using the Serverless Framework, you can use the power tuning plugin at https://github.com/alexcasalboni/aws-lambda-power-tuning to help estimate your function's resource usage.

Resource allocation during cold starts

AWS Lambda respects the resource allocation while executing your function but will attempt to "boost" the CPU available while loading and initializing your function dependencies. This means that increasing the resource allocation will not really make a difference to your cold starts.

12.3.3 Optimize function logic

AWS Lambda bills your usage based on the time your function starts executing to the time it stops executing – not by CPU cycles spent or any other time-based metrics. This implies that what your function does during that is important. Consider the image-resizer-service function: when you are downloading the S3 object, your code is simply waiting for S3 service to respond, and you are paying for that wait time. In this function's case the time spent is negligible, but this wait time can get excessive for services that have long response times (for example, waiting on an EC2 instance being provisioned) or wait times (such as downloading a very large file). There are two options to minimize this idle time:

1. *Minimize orchestration in code* – Instead of waiting on an operation inside your function, use AWS Step functions to separate the "before" and "After" logic as two separate functions. For example, if you have logic that needs to run before and after an API call is made, sequence them as two separate functions and use an AWS Step function to orchestrate between them.
2. *Use threads for I/O intensive operations* – You can use multiple threads within a Lambda function (if the programming language supports it), just like code running in any compute environment. However, unlike conventional programs, the best use for multi-threading isn't for parallelizing computations. This is because Lambda does not allocate multiple cores to Lambda functions running with memory less than 1.8GB, so you need to allocate more resources to get the parallelization benefit. Instead, you can use threads as a way to parallelize I/O operations. For example, a python version of the image_resizer function could act on multiple functions by executing the S3 download on a separate thread to thumbnailing.

By following these best practices, you can significantly reduce the latency (and cost!) of your serverless application.

12.4 Concurrency

Another important concept to understand for AWS Lambda functions is concurrency. *Concurrency* is the unit of scale for a Lambda function. Underneath the covers, it maps to the number of execution environments assigned to requests. You can estimate the concurrency of your function at any time with the following formula:

```
Concurrency = Requests per second (TPS) * function duration
```

Using peak values will give you peak concurrency; using average values will give you average concurrency. You can monitor the concurrency for any given function (and for the overall account) using the ConcurrentExecutions CloudWatch metric.

AWS Lambda enforces two limits to the concurrency of a function.

- There is an account-wide soft limit on the total concurrent executions of all functions within the account. This is set by default to 1000 at the time of writing, and it can be raised to desired values through a support case. You can view the account-level setting by using the GetAccountSettings API and viewing the AccountLimit object.
- There is also an account-wide limit on the rate at which you can scale up your

concurrent executions. In larger AWS regions, you are allowed to scale instantly to 3000 concurrent and then add 500 concurrent executions every subsequent minute; this limit is lower in smaller regions. These limits may change, so be sure to refer to the latest values listed in https://docs.aws.amazon.com/lambda/latest/dg/scaling.html#scaling-behavior.

This makes it important to always estimate what your peak and average concurrency needs will be, how quickly you'll need to ramp up, and file a request to raise limits as needed.

12.4.1 Correlation between Requests, latency, and concurrency

For most functions, concurrency increases as a function of requests and function duration, subject to the concurrency limits on the function and account. However, for functions used to process stream data (Kinesis and DynamoDB streams), the concurrency is determined by the number of shards on the stream being processed. Given that latency is determined by the function itself, this means for stream-processing functions, you may see variable request rate or throughput. To put it another way,

```
Effective Processing rate : Effective concurrency / average duration (events per second)
```

Consider a function that takes 1 second to process a stream with 5 shards, with a batch size of 100. This means the maximum number of requests (each with 100 records) that the function can process would be 5, and the maximum number of records processed at any given time would be 5*100=500. On the other hand, if the same stream had 10 shards, the throughput would double as well.

12.4.2 Managing concurrency

AWS offers two settings for managing concurrency. The first one is the account level concurrency limit, that is enforced on the total concurrency across all functions within your account. This limit is set to 1000 by default and can be raised through a service limit increase ticket – you cannot "Self-service" this increase. The second is a per-function concurrency control, which you can use to control the concurrency of an individual function. You only use the per function concurrency control if you have a function that you want to "reserve" concurrency for, or a function that needs to be limited in its concurrency (because of a downstream resource). For example, you may want to restrict how high a Lambda function scales because it calls an API that can only handle a certain load. If this was left unchecked, then your function could cause the downstream API to be overloaded, causing an availability for your overall application. This makes monitoring concurrency and managing it an important step to follow. You can learn more about the limits and the controls here https://docs.aws.amazon.com/lambda/latest/dg/concurrent-executions.html.

12.5 Summary

- Serverless applications do not require conventional application performance monitoring steps. Instead, optimizing the performance of your function code gives you the most gain.
- Leverage the toolsets (like X-Ray) and configurations (like the memory setting) to easily

locate and optimize performance.

- Concurrency for Lambda functions can affect your function latency (and vice versa), so ensure you monitor and manage it for your critical functions.

A

Services for your serverless architecture

AWS is a giant playground of different services and products you can use to build serverless applications. Lambda is a key service we'll discuss in this book, but other services and products can be just as useful, if not crucial, for solving certain problems. There are many excellent non-AWS products too, so don't feel obligated to use only what Amazon has to offer. Have a look at the offerings from Microsoft and Google too.

The following is a sample of services that we've found useful. You can use this appendix as a guide to various services and products we'll discuss throughout the book.

A.1 API Gateway

The Amazon API Gateway is a service that you can use to create an API layer between the front-end and back-end services. The lifecycle management of the API Gateway allows multiple versions of the API to be run at the same time, and it supports multiple release stages such as development, staging, and production. API Gateway also comes with useful features like caching and throttling of requests.

The API is defined around resources and methods. A resource is a logical entity such as a user or product. A method is a combination of an HTTP verb, such as GET, POST, PUT, or DELETE, and the resource path. API Gateway integrates with Lambda and other AWS services. It can be used as a proxy service and forward requests to regular HTTP endpoints.

A.2 Simple Notification Service

Amazon Simple Notification Service (SNS) is a scalable pub-sub service designed to deliver messages. Producers or publishers create and send messages to a topic. Subscribers or consumers subscribe to a topic and receive messages over one of the supported protocols. SNS stores messages across multiple servers and data centers for redundancy and guarantees at-least-once delivery. At-least-once delivery stipulates that a message will be delivered at

least once to a subscriber but, on rare occasions due to the distributed nature of SNS, it may be delivered multiple times.

In cases when a message can't be delivered by SNS to HTTP endpoints, it can be configured to retry deliveries at a later time. SNS can also retry failed deliveries to Lambda in cases where throttling is applied. SNS supports message payloads of up to 256 KB.

A.3 Simple Storage Service

Simple Storage Service (S3) is Amazon's scalable storage solution. Data in S3 is stored redundantly across multiple facilities and servers. The event notifications system allows S3 to send events to SNS, SQS, or Lambda when objects are created or deleted. S3 is secure by default, with only owners having access to the resources they create, but it's possible to set more granular and flexible access permissions using access control lists and bucket policies.

S3 uses the concept of buckets and objects. *Buckets* are high-level directories or containers for objects. *Objects* are a combination of data, metadata, and a key. A *key* is a unique identifier for an object in a bucket. S3 also supports the concept of a *folder* as a means of grouping objects in the S3 console. Folders work by using key name prefixes. A forward slash character "/" in the key name delineates a folder. For example, an object with the key name documents/personal/myfile.txt is represented as a folder called documents, containing a folder called personal, containing the file myfile.txt in the S3 console.

A.4 Simple Queue Service

Simple Queue Service (SQS) is Amazon's distributed and fault-tolerant queuing service. It ensures at-least-once delivery of messages similar to SNS and supports message payloads of up to 256 KB. SQS allows multiple publishers and consumers to interact with the same queue, and it has a built-in message lifecycle that automatically expires and deletes messages after a preset retention period. As with most AWS products, there are access controls to help control access to the queue. SQS integrates with SNS to automatically receive and queue messages.

A.5 Simple Email Service

Simple Email Service (SES) is a service designed to send and receive email. SES handles email-receiving operations such as scanning for spam and viruses and rejection of email from untrusted sources. Incoming email can be delivered to an S3 bucket, or used to invoke a Lambda notification or create an SNS notification. These actions can be configured as part of the receipt rule, which tells SES what to do with the email once it arrives.

Sending emails with SES is straightforward but there are limits, which are in place to regulate the rate and the number of messages being sent out. SES will automatically increase the quota as long as high-quality email, and not spam, is being sent.

A.6 Relational Database Service

Amazon Relational Database Service (RDS) is a web service that helps with the setup and operation of a relational database in the AWS infrastructure. RDS supports the Amazon Aurora, MySQL, MariaDB, Oracle, MS-SQL, and PostgreSQL database engines. It takes care of routine tasks such as provisioning, backup, patching, recovery, repair, and failure detection. Monitoring and metrics, database snapshots, and multiple availability zone (AZ) support are

provided out of the box. RDS uses SNS to deliver notifications when an event occurs. This makes it easy to respond to database events such as creation, deletion, failover, recovery, and restoration when they happen.

A.7 DynamoDB

DynamoDB is Amazon's NoSQL database. Tables, items, and attributes are Dynamo's main concepts. A table stores a collection of items. An item is made up of a collection of attributes. Each attribute is a simple piece of data such as a person's name or phone number. Every item is uniquely identifiable. Lambda integrates with DynamoDB tables and can be triggered by a table update. Global Tables is a notable feature of Dynamo that seamlessly replicates tables across different AWS regions and resolves any data conflicts (using "last writer wins" reconciliation to handle concurrent updates). It makes DynamoDB a good database for scalable, global applications. Finally, an in-memory cache (DAX) is available for DynamoDB. It shortens the response time but comes at a price.

A.8 Algolia

Algolia is a (non-AWS) managed search engine API. It can search through semi-structured data and has APIs to allow developers to integrate search directly into their websites and mobile applications. One of Algolia's outstanding capabilities is its speed. Algolia can distribute and synchronize data across 15 regions around the world and direct queries to the closest data center. Algolia has a concept of indices ("an entity where you import the data you want to search.. analogous to a table within a database"), records ("a JSON schemaless object that you want to be searchable") and operations, which are essentially atomic actions such as update or delete. These concepts are straightforward and make Algolia one of the easier search platforms to use. Paid plans begin from about $35 per month but can quickly grow in cost, depending on the number of records and operations performed by your application and users.

A.9 Media Services

AWS Media Services are a new product designed for developers to build video workflows. Media Services consist of the following products:

- MediaConvert is designed to transcode between different video formats at scale.
- MediaLive is a live video-processing service. It takes a live video source and compresses it into smaller versions for distribution.
- MediaPackage enables developers to implement video features such as pause and rewind. It can also be used add Digital Right Management (DRM) to content.
- MediaStore is a storage service optimized for media. Its aim is to provide a low-latency storage system for live and on-demand video content.
- MediaTailor enables developers to insert individually targeted ads in to the video stream.

Media Services provide an advanced suite of services that are superior to Elastic Transcoder. Nevertheless, Elastic Transcoder has a few features – such as the ability to create WebM files and animated GIFs—that Media Services is missing.

A.10 Kinesis Streams

Kinesis Streams is a service for real-time processing of streaming big data. It's typically used for quick log and data intake, metrics, analytics, and reporting. It's different from SQS in that Amazon recommends that Kinesis Streams be used primarily for streaming big data, whereas SQS is used as a reliable hosted queue, especially if more fine-grained control over messages, such as visibility timeouts or individual delays, is required.

In Kinesis Streams, shards specify the throughput capacity of a stream. The number of shards needs to be stipulated when the stream is created, but resharding is possible if throughput needs to be increased or reduced. In comparison, SQS makes scaling much more transparent. Lambda can integrate with Kinesis to read batches of records from a stream as soon as they're detected.

A.11 Athena

AWS bills Athena as a serverless interactive query service. Essentially, this service allows you to query data placed into S3 using standard SQL. In a lot of cases, there's no need to run ETL (Extract, Transform, and Load) jobs to transform your data before querying can take place (although you can combine Athena with AWS Glue if you needed to transform your data to be a certain way). As a user, you upload data to S3, prepare a schema, and begin querying almost immediately.

A.12 AppSync

AppSync is billed as allowing developers to create "data driven apps with real-time and offline capabilities." In reality, AppSync is a managed GraphQL endpoint provided by AWS. It integrates with DynamoDB, Lambda, and Amazon Elasticsearch. If you are familiar with GraphQL and GraphQL schemas, you can get started with AppSync straight away. If you are not familiar with GraphQL, we recommend doing a bit of reading beforehand (http://graphql.org/learn/). GraphQL has certainly been finding its share of acclaim over the past few years, particularly among adopters of serverless technologies.

A.13 Cognito

Amazon Cognito is an identity management service. It integrates with public identity providers such as Google, Facebook, Twitter, and Amazon or with your own system. Cognito supports user pools, which allow you to create your own user directory. This allows you to register and authenticate users without having to run a separate user database and authentication service. Cognito supports synchronization of user application data across different devices and has offline support that allows mobile devices to function even when there's no internet access.

A.14 Auth0

Auth0 is a non-AWS identity management product that has a few features that Cognito doesn't. Auth0 integrates with more than 30 identity providers, including Google, Facebook, Twitter, Amazon, LinkedIn, and Windows Live. It provides a way to register new users through the use of its own user database, without having to integrate with an identity provider. In addition, it has a facility to import users from other databases.

As expected, Auth0 supports standard industry protocols including SAML, OpenID Connect, OAuth 2.0, OAuth 1.0, and JSON Web Token. It's dead simple to integrate with AWS Identity and Access Management and with Cognito.

A.15 Other services

The list of services provided in this section is a short sample of the different products you can use to build your application. There are many more services, including those provided by large cloud-focused companies such as Google and Microsoft and smaller, independent companies like Auth0.

There are also auxiliary services that you need to be aware of. These can help you be more efficient and build software faster, improve performance, or achieve other goals. When building software, consider the following products and services:

- Content Delivery Networks (CloudFront, CloudFlare)
- DNS management (Route 53)
- Caching (ElastiCache)
- Source control (GitHub, GitLab)
- Continuous integration and deployment (Circle CI, GitHub Actions)

For every service suggestion, you can find alternatives that may be just as good or even better, depending on your circumstances. We urge you to do more research and explore the various services that are currently available.

B

Setting up your cloud

Most of the architecture described in this book is built on top of AWS. This means you need a clear understanding of AWS from the perspectives of security, alerting, and costs. It doesn't matter whether you use Lambda alone or have a large mix of services. Being able to configure security, knowing how to set up alerts, and controlling cost are important. This appendix is designed so that you can understand these concerns and learn where to look for important information in AWS.

AWS security is a complex subject, but this appendix gives you an overview of the difference between users and roles and shows you how to create policies. This information is needed to configure a system in which services can communicate effectively and securely. Some of the time you will not need to create or configure policies directly – tools like Serverless framework will do it for you. But it's still important to understand how the pieces fit together and where to look for help if things go wrong.

Cost is an important consideration when using a platform such as AWS and implementing serverless architecture. It's essential to understand the cost calculation of the services you're going to use. This is useful not only for avoiding bill shock but also for predicting next month's bill and beyond. We look at estimating the cost of services and discuss strategies for tracking costs and keeping them under control. This appendix is not an exhaustive guide to AWS. If you have further questions after reading this appendix, take a look at AWS documentation (https://aws.amazon.com/documentation).

B.1 Security model and identity management

In chapter 2 you created an Identity and Access Management (IAM) user and a number of roles in order to use Lambda, S3, and MediaConvert. In this section, you'll take your new-found knowledge and develop it further by learning about users, groups, roles, and policies in more detail.

B.1.1 Creating and managing IAM users

As you'll recall, an IAM user is an entity in AWS that identifies a human user, an application, or a service. A user normally has a set of credentials and permissions that can be used to access resources and services across AWS.

An IAM user typically has a friendly name to help you identify the user and an Amazon Resource Name (ARN) that uniquely identifies it across AWS. Figure B.1 shows a summary page and an ARN for a fictional user named Alfred. You can get to this summary in the AWS console by clicking IAM in the AWS console, clicking Users in the navigation pane, and then clicking the name of the user you want to view.

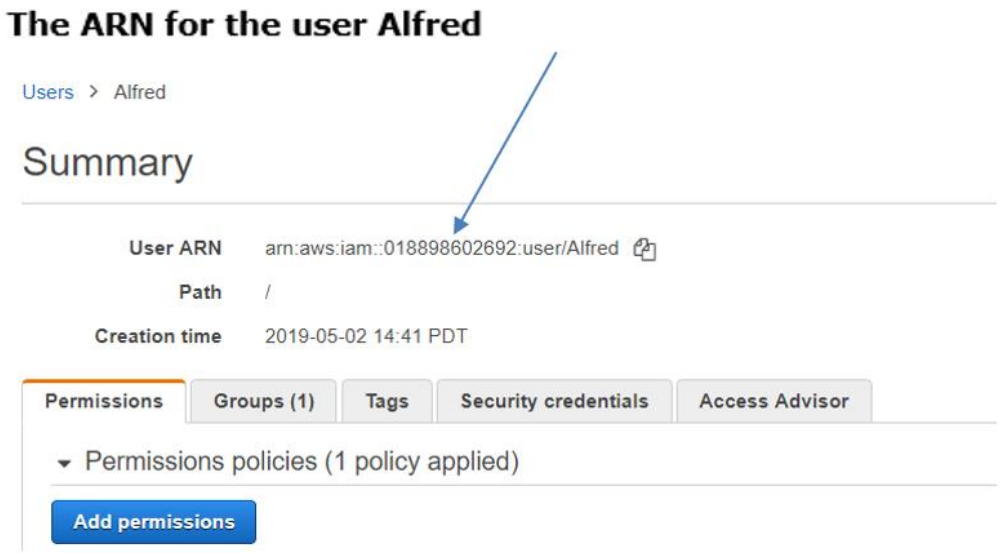


Figure B.1 The IAM console shows metadata such as the ARN, groups, and creation time for every IAM user in your account.

You can create IAM users to represent human users, applications, or services. IAM users created to work on behalf of an application or a service sometimes are referred to as service accounts. These types of IAM users can access AWS service APIs using an access key. An access key for an IAM user can be generated when the user is initially created, or you can create it later by clicking Users in the IAM console, clicking the required user name, selecting Security Credentials, and then clicking the Create Access Key button.

The two components of an access key are the Access Key ID and the Secret Access Key. The Access Key ID can be shared publicly, but the Secret Access Key must be kept hidden. If the Secret Access Key is revealed, the whole key must be immediately invalidated and re-created. An IAM user can have, at most, two active access keys.

If an IAM user is created for a real person, then that user should be assigned a password. This password will allow a human user to log into the AWS console and use services and APIs directly.

To create a password for an IAM user, follow these steps:

1. In the IAM console, click Users in the navigation pane.
2. Click the required username to open the user's settings.
3. Click the Security Credentials tab and then click Manage Password (figure B.2).

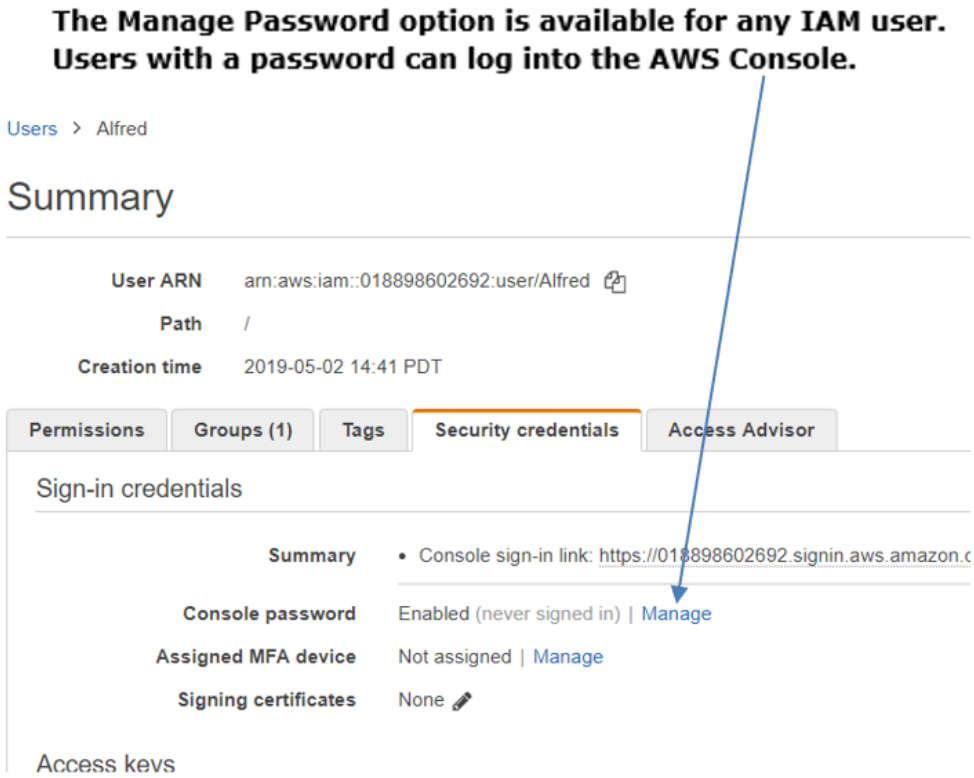


Figure B.2 IAM users have a number of options, including being able to set a password, change access keys, and enable multifactor authentication.

4. In the popup, choose whether to enable or disable console access, type in a new custom password, or let the system autogenerate one. You can also force the user to create a new password at the next sign-in.

In the popup, choose whether to enable or disable console access, type in a new custom password, or let the system autogenerate one. You can also force the user to create a new password at the next sign-in (figure B.3).

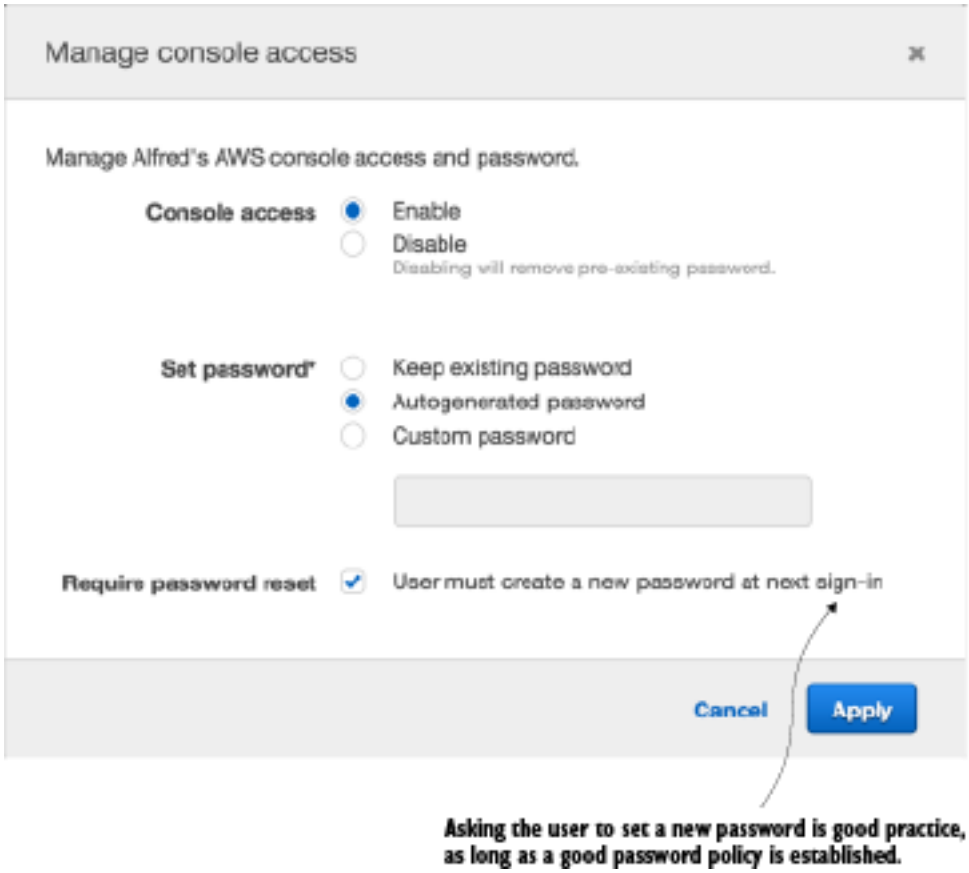


Figure B.3 Make sure to create a good password policy with a high degree of complexity if you allow users to log into the AWS console. Password policy can be set up in Account Settings of the IAM console.

After a user is assigned a password, they can log into the AWS console by navigating to https://<Account-ID>.signin.aws.amazon.com/console. To get the account ID, click Support in the upper-right navigation bar, and then click Support Center. The account ID (or account number) is shown at the top of the console. You may want to set up an alias for the account ID also, so that your users don't have to remember it (for more information about aliases, see http://amzn.to/1MgvWvf).

Multi-factor authentication

Multi-factor authentication (MFA) adds another layer of security by prompting users to enter an authentication code from their MFA device when they try to sign into the console (this is in addition to the usual username and password). It makes it more difficult for an attacker to compromise an account. Any modern smartphone can act as a virtual MFA appliance using an application such as Google Authenticator or AWS Virtual MFA. It's recommended that you enable MFA for any user who might use the AWS console. You'll find the option Assign MFA Device in the Security Credentials tab when you click an IAM user in the console.

Temporary security credentials

At this time, there's a limit of 5,000 users per AWS account, but you can raise the limit if needed. An alternative to increasing the number of users is to use temporary security credentials. Temporary Security Credentials can be set up to expire after a short and can be generated dynamically. See Amazon's online documentation at http://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_temp.html for more information on temporary security credentials. You can find more information about IAM users at http://docs.aws.amazon.com/IAM/latest/UserGuide/id_users.html.

B.1.2 Groups

Groups represent a collection of IAM users. They provide an easy way to specify permissions for multiple users at once. For example, you may want to create a group for developers or testers in your organization or have a group called Lambda to allow all members of that group to execute Lambda functions. Amazon recommends using groups to assign permissions to IAM users rather than defining permissions individually.

Any user who joins a group inherits permissions assigned to the group. Similarly, if a user leaves a group, the group's permissions are removed from the user. Furthermore, groups can contain only users, not other groups or entities such as roles.

B.1.3 Roles

A *role* is a set of permissions that a user, application, or a service can assume for a period of time. A role is not uniquely coupled to a specific user, nor does it have associated credentials such as passwords or access keys. It's designed to grant permissions to a user or a service that typically doesn't have access to the required resource.

Delegation is an important concept associated with roles. Put simply, delegation is concerned with the granting of permissions to a third party to allow access to a particular resource. It involves establishing a trust relationship between a trusting account that owns the resource and a trusted account that contains the users or applications that need to access the resource. Figure B.4 shows a role with a trust relationship established for a service called CloudCheckr, which you can read more about in section B.3.2.

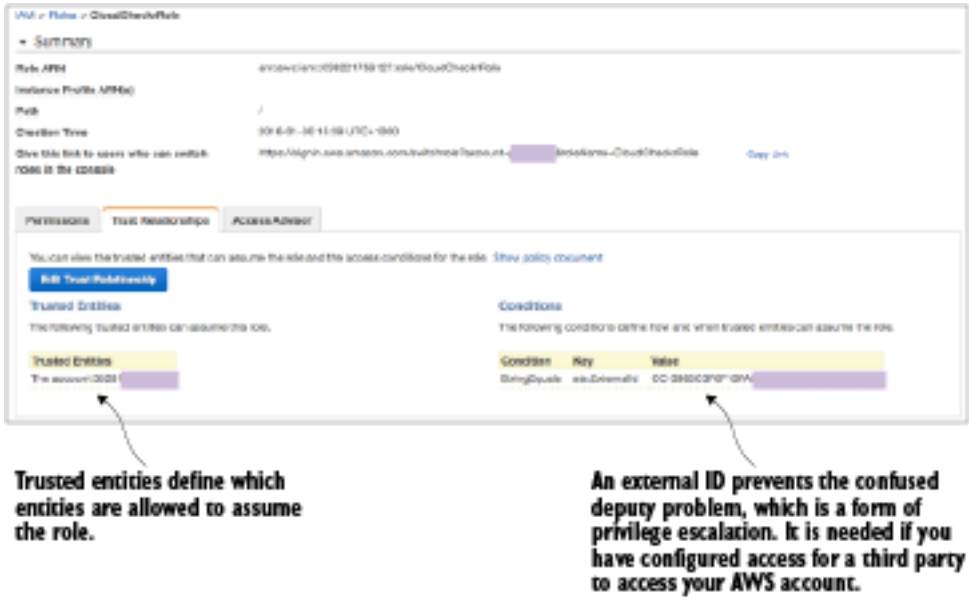


Figure B.4 This role grants CloudCheckr access to the AWS account to perform analysis of costs and recommend improvements.

Federation is another concept that's discussed often in the context of roles. Federation is the process of creating a trust relationship between an external identity provider such as Facebook, Google, or an enterprise identity system that supports Security Assertion Markup Language (SAML) 2.0, and AWS. It enables users to log in via one of those external identity providers and assume an IAM role with temporary credentials.

B.1.4 Resources

Permissions in AWS are either identity-based or resource-based. Identity-based permissions specify what an IAM user or a role may do. Resource-based permissions specify what an AWS resource, such as an S3 bucket or an SNS topic, is allowed to do or who can have access to it.

A resource-based policy often specifies who has access to the given resource. This allows trusted users to access the resource without having to assume a role. The AWS user guide states: "cross-account access with a resource-based policy has an advantage over a role. With a resource that is accessed through a resource-based policy, the user still works in the trusted account and does not have to give up his or her user permissions in place of the role permissions. In other words, the user continues to have access to resources in the trusted account at the same time as he or she has access to the resource in the trusting account" (http://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_compare-resource-policies.html). Not all AWS services support resource-based policies (the user guide at https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_aws-services-that-work-with-iam.html lists all the services that do).

B.1.5 Permissions and policies

When you initially create an IAM user, it's not able to access or do anything in your account. You need to grant the user permissions by creating a policy that describes what the user is allowed to do. The same goes for a new group or role. A new group or a role needs to be assigned a policy to have any effect.

The scope of any policy can vary. You can give your user or role administrator access to the whole account or specify individual actions. It's better to be granular and specify only permissions that are needed to get the job done (least privilege access). Start with a minimum set of permissions and add additional permissions only if necessary.

There are two types of policies: managed and inline. Managed policies apply to users, groups, and roles but not to resources. Managed policies are standalone. Some managed policies are created and maintained by AWS. You also can also create and maintain customer-managed policies. Managed policies are great for reusability and change management. If you use a customer-managed policy and decide to modify it, all changes are automatically applied to all IAM users, roles, and groups that the policy is attached to. Managed policies allow for easier versioning and rollbacks.

Inline policies are created and attached directly to a specific user, group, or role. When an entity is deleted, the inline policies embedded within it are deleted also. Resource-based policies are always inline. To add an inline or a managed policy, click into the required user, group, or role and then click the Permissions tab. You can attach, view, or detach a managed policy and similarly create, view, or remove an inline policy.

A policy is specified using JSON notation. The following listing shows a managed `AWSLambdaExecute` policy.

Listing B.1 `AWSLambdaExecute` **policy**

```
{
   "Version":"2012-10-17",                                    #A
   "Statement":[                                              #B
      {
         "Effect":"Allow",
         "Action": "logs:*",
         "Resource":"arn:aws:logs:*:*:*"
      },
      {
         "Effect":"Allow",                                    #C
         "Action":[                                           #D
            "s3:GetObject",
            "s3:PutObject"
         ],
         "Resource":"arn:aws:s3:::*"                          #E
      }
   ]
}
```

#A Version specifies the policy language version. The current version is 2012-10-17. If you're creating a custom policy, make sure to include the version and set it to 2012-10-17.

#B The statement array contains one or more statements that specify the actual permissions that make up the policy.

#C The Effect element is required and specifies whether the statement allows or denies access to the resource. The only two available options are Allow and Deny.

#D The Action element, or an array, specifies the specific actions on the resource that should be allowed or denied. The use of a wildcard (*) character is allowed; for example, "Action": "s3:*".
#E The Resource element identifies the object or objects that the statement applies to. It can be specific or include a wildcard to refer to multiple entities.

Many IAM policies contain additional elements such as Principal, Sid, and Condition. The Principal element specifies an IAM user, an account, or a service that's allowed or denied access to a resource. The Principal element isn't used in policies that are attached to IAM users or groups. Instead, they're used in roles to specify who can assume the role. They're also common to resource-based policies. Statement ID (Sid) is required in policies for certain AWS services, such as SNS. A condition allows you to specify rules that dictate when a policy should apply. An example of a condition is presented in the next listing.

Listing B.2 Policy condition

```
"Condition": {
   "DateLessThan": {                                    #A
              "aws:CurrentTime": "2020-09-12T12:00:00Z"
        },
        "IpAddress": {
               "aws:SourceIp": "127.0.0.1"            #B
        }
   }
```

#A You can use a number of conditional elements. These include DateEquals, DateLessThan, DateMoreThan, StringEquals, StringLike, StringNotEquals, and ArnEquals.
#B The condition keys represent values that come from the request issued by a user. Possible keys include SourceIp, CurrentTime, Referer, SourceArn, userid, and username. The value can be either a specific literal value such as "127.0.0.1" or a policy variable.

Multiple conditions

The AWS documentation at http://amzn.to/21UofNi states "If there are multiple condition operators, or if there are multiple keys attached to a single condition operator, the conditions are evaluated using a logical AND. If a single condition operator includes multiple values for one key, that condition operator is evaluated using a logical OR." See http://amzn.to/21UofNi for great examples you can follow and a whole heap of useful documentation.

Amazon recommends using conditions, to the extent that is practical, for security. The next listing, for example, shows an S3 bucket policy that forces content to be served only over HTTPS/SSL. This policy refuses connections over unencrypted HTTP.

Listing B.3 Policy to enforce HTTPS/SSL

```
{
    "Version": "2012-10-17",
    "Id": "123",
    "Statement": [
        {
            "Effect": "Deny",                       #A
            "Principal": "*",
            "Action": "s3:*",                       #A
            "Resource": "arn:aws:s3:::my-bucket/*",
```

```
            "Condition": {
                "Bool": {
                    "aws:SecureTransport": false   #B
                }
            }
        }
    ]
}
```

#A This policy explicitly denies access to s3 if the condition is met.
#B The condition is met when requests are not sent using SSL. This forces the policy to block access to the bucket if a user tries to access it over regular, unencrypted HTTP.

B.2 AWS CloudTrail

CloudTrail is an AWS service that records API calls. This service records information such as the identity of the API caller, the source IP address, and the event. This data is saved in a log file in an S3 bucket. CloudTrail is an effective way to generate logs and gather information about what AWS services are doing and who's invoking them. For example, you can use it to find who and when deleted a useful S3 bucket. CloudTrail supports a number of AWS services including DynamoDB, Kinesis, API Gateway, and Lambda, and it can be configured to push logs straight to a CloudWatch log group.

The free tier in CloudTrail allows you to create a free trail per region. For additional trails, however, there is a charge. AWS makes a distinction between *Management events* and *Data events*. Management events are operations performed on resources (such as creating an S3 bucket). AWS charges these at $2.00 per 100,000 events recorded. A Data event is an operation performed on or within a resource. An example of this is Lambda being invoked using its API. Data events are charged at $0.10 per 100,000 events. A more detailed pricing description as well as examples can be found at https://aws.amazon.com/cloudtrail/pricing/.

B.2.1 Enabling CloudTrail

CloudTrail records API calls made across your account by users or services on behalf of a user. It provides a convenient way to audit API calls and to help diagnose and troubleshoot issues. CloudTrail introduces the concept of trails, which are configurations that enable logging of APIs. CloudTrail, as we discussed at the beginning of section B.2, can log Management Events (operations performed on resources) and Data Events (events performed within a resource such as S3 and Lambda). You should enable CloudTrail because it helps to understand what's happening within the system when things go wrong. Let's walk through the steps to create a trail for your region:

1. Click CloudTrail in the AWS console, and then click Get Started Now.
2. Give your trail a name, such as 24-Hour-Video, and set Apply Trail to All Regions to No.
3. Leave Read/Write events on All. We are going to track all Management events.
4. Click the Lambda tab and select the functions that you created in chapter 2. We are also going to record Lambda Invoke API operations.

5. Check that Create a New S3 Bucket is set to Yes.
6. Type in a name for your new CloudTrail S3 bucket.
7. Click Advanced to have a look at additional options. You don't need to set anything, although you could enable SNS notifications if you wanted to (figure B.5).

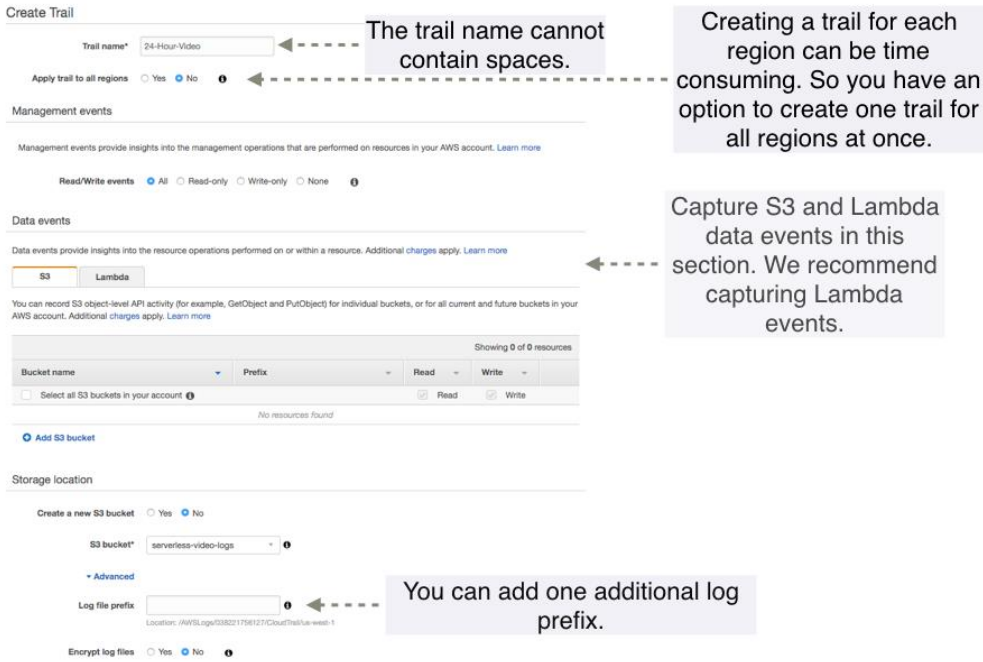


Figure B.5 Creating a new trail shouldn't take long. The only two mandatory fields are the name and the S3 bucket.

8. Click Create to save the trail and complete your configuration.
9. After the trail is created, you'll be transferred to a table that lists all trails for your account across all regions.

The API Activity History page in CloudTrail shows create, modify, and delete API calls executed over the past seven days. You can expand each event and then click the View Event button to see more information. Events may take up to 15 minutes to appear.

Whether you decide to use CloudTrail is up to you but it's important to know about this service particularly as you begin to mature your systems running on AWS.

B.3 Cost

Receiving an unpleasant surprise in the form of a large bill at the end of the month is disappointing and stressful. Amazon CloudWatch can create billing alarms that send notifications if total charges for the month exceed a predefined threshold. This is useful not only to avoid unexpectedly large bills but also to catch potential misconfigurations of your system. For example, it's easy to misconfigure a Lambda function and inadvertently allocate 3.0 GB of RAM to it. The function might not do anything useful except wait for 15 seconds to receive a response from a database. In a very heavy-duty environment, the system might perform 2 million invocations of the function a month, costing a little over $1462. The same function with 128 MB of RAM would cost around $56 per month. If you perform cost calculations up front and have a sensible billing alarm, you'll quickly realize that something is going on when billing alerts begin to come through.

B.3.1 Creating billing alerts

Follow these steps to create a billing alert:

1. In the main AWS console, click your name (or the name of the IAM user that's representing you) and then click My Billing Dashboard.
2. Click Preferences in the navigation pane and then enable the check box next to Receive Billing Alerts.
3. Click Save preferences.
4. Go back to the main AWS console and find the CloudWatch service.
5. Open the CloudWatch service and select Billing in the navigation pane.
6. Click the Create Alarm button and then click the Billing Metrics subheading.
7. Under Billing > Total Estimated Charge select the first check box (this is the Estimated Charges metric). Selecting this option captures estimated charges across all AWS services. You can, however, go granular and select specific services.
8. Click the Create Alarm button in the lower-right corner to open the Create Alarm dialog.
9. Set your spending threshold and select an SNS topic for the delivery of notifications. You can, optionally, click New list and enter an email address directly (figure B.6).

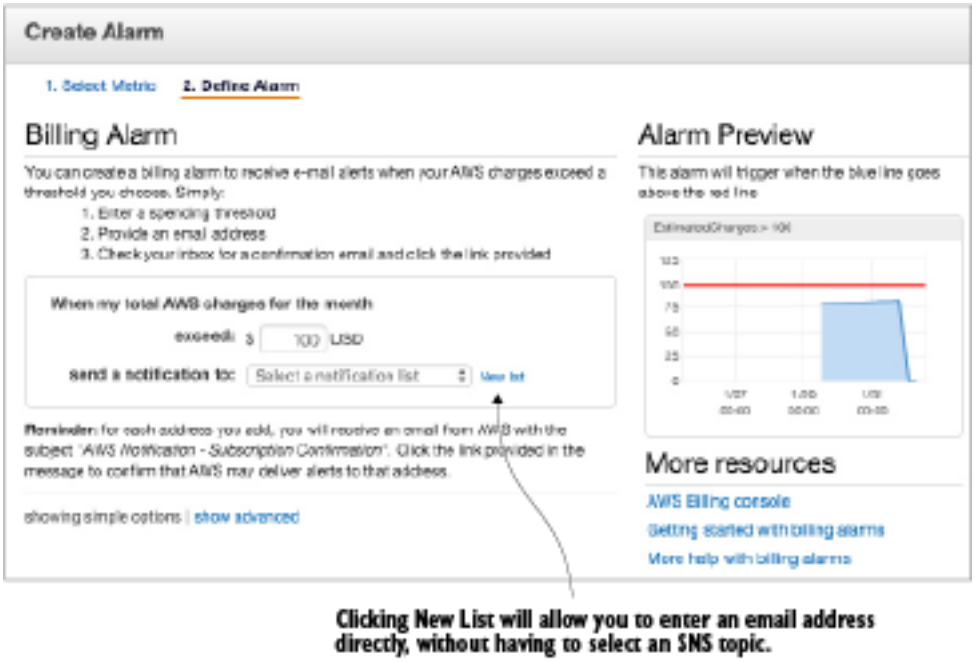


Figure B.6 It's good practice to create multiple billing alarms to keep you informed of ongoing costs.

B.3.2 Monitoring and optimizing costs

Services such as CloudCheckr (http://cloudcheckr.com) can help to track costs, send alerts, and even suggest savings by analyzing services and resources in use. CloudCheckr comprises several different AWS services, including S3, CloudSearch, SES, SNS, and DynamoDB. It's richer in features and easier to use than some of the standard AWS features. It's worth considering for its recommendations and daily notifications.

AWS also has a service called Trusted Advisor that suggests improvements to performance, fault tolerance, security, and cost optimization. Unfortunately, the free version of Trusted Advisor is limited, so if you want to explore all of the features and recommendations it has to offer, you must upgrade to a paid monthly plan or access it through an AWS enterprise account.

Cost Explorer (figure B.7) is a useful, albeit high-level, reporting and analytics tool built into AWS. You must activate it first by clicking your name (or the IAM username) in the top-right corner of the AWS console, selecting My Billing Dashboard, then clicking Cost Explorer from the navigation pane and enabling it. Cost Explorer analyzes your costs for the current month and the past four months. It then creates a forecast for the next three months. Initially, you may not see any information, because it takes 24 hours for AWS to process data for the current month. Processing data for previous months make take even longer. More information about Cost Explorer is available at http://amzn.to/1KvN0g2.

Figure B.7 The Cost Explorer tool allows you to review historical costs and estimate what future costs may be.

B.3.3 Using the Simple Monthly Calculator

The AWS Pricing Calculator (https://calculator.aws) is a web application developed by Amazon to help model costs for many of its services. This tool allows you to select a service, enter information related to the consumption of that particular resource and get an indicative cost.

B.3.4 Calculating Lambda and API Gateway costs

The cost of running serverless architecture often can be a lot less than running traditional infrastructure. Naturally, the cost of each service you might use will be different, but you can look at what it takes to run a serverless system with Lambda and the API Gateway.

Amazon's pricing for Lambda (https://aws.amazon.com/lambda/pricing/) is based on the number of requests, duration of execution, and the amount of memory allocated to the function. The first million requests are free, with each subsequent million charged at $0.20. Duration is based on how long the function takes to execute, rounded up to the next 100ms. Amazon charges in 100ms increments while also factoring in the amount of memory reserved for the function.

A function created with 1 GB of memory will cost $0.000001667 per 100ms of execution time, whereas a function created with 128 MB of memory will cost $0.000000208 per 100ms. Note that Amazon prices may differ depending on the region and that they're subject to change at any time.

Amazon provides a perpetual free tier with 1 million free requests and 400,000 GB-seconds of compute time per month. This means that a user can perform a million requests and spend an equivalent of 400,000 seconds running a function created with 1 GB of memory before they have to pay.

As an example, consider a scenario where you have to run a 256 MB function 5 million times a month. The function executes for two seconds each time. The cost calculation follows:

Monthly request charge:

- The free tier provides 1 million requests, which means that there are only 4 million billable requests (5M requests – 1M free requests = 4M requests).
- Each million is priced at $0.20, which makes the request charge $0.80 (4M requests * $0.2/M = $0.80).

Monthly compute charge:

- The compute price for a function per GB-second is $0.00001667. The free tier provides 400,000 GB-seconds free.
- In this scenario, the function runs for 10ms (5M * 2 seconds).
- 10M seconds at 256 MB of memory equates to 2,500,000 GB-seconds (10,000,000 * 256 MB/1024 = 2,500,000).
- The total billable amount of GB-seconds for the month is 2,100,000 (2,500,000 GB-seconds – 400,000 free tier GB-seconds = 2,100,000).
- The compute charge is therefore $35.007 (2,100,000 GB-seconds * $0.00001667 = $35.007).

The total cost of running Lambda in this example is $35.807. The API Gateway pricing is based on the number of API calls received and the amount of data transferred out of AWS. In the eastern United States, Amazon charges $3.50 for each million API calls received and $0.09/GB for the first 10 TB transferred out. Given the previous example and assuming that monthly outbound data transfer is 100 GB a month, the API Gateway pricing is as follows:

Monthly API charge:

- The free tier includes one million API calls per month but is valid for only 12 months. Given that it's not a perpetual free tier, it won't be included in this calculation.
- The total API cost is $17.50 (5M requests * $3.50/M = $17.50).

Monthly data charge:

- The data charge is $9.00 (100 GB * $0.09/GB = $9).

The API Gateway cost in this example is $26.50. The total cost of Lambda and the API Gateway is $62.307 per month. It's worthwhile to attempt to model how many requests and operations you may have to handle on an ongoing basis. If you expect 2M invocations of a Lambda function that uses only 128 MB of memory and runs for a second, you'll pay approximately $0.20 month. If you expect 2M invocations of a function with 512 MB of RAM that runs for 5 seconds, you'll pay a little more than $75.00. With Lambda, you have an opportunity to assess costs, plan ahead, and pay for only what you actually use. Finally, don't forget to factor in other services such as S3 or SNS, no matter how insignificant their cost may seem to be.

Serverless calculator

The online serverless calculator (http://serverlesscalc.com) is an easy-to-use tool we built to help you model Lambda costs. All you need to do is specify the number of Lambda executions per month, the estimated execution time, and the size (in memory) of your function. The calculator will immediately show you the monthly Lambda charge, including request and compute cost breakdowns. Moreover, this tool will allow you to compare Lambda costs to other similar serverless compute technologies such as Azure Functions, Google Cloud Functions, and IBM Cloud Functions.

B.4 Summary

- Identity and Access Management in AWS includes users, groups, roles, policies, and permissions
- You can monitor ongoing costs using built-in alerting and services such as CloudCheckr and Trusted Advisor
- You can set up CloudTrail to monitor API invocations of AWS services
- You can estimate costs for Lambda and the API Gateway and using the Simple Monthly Calculator