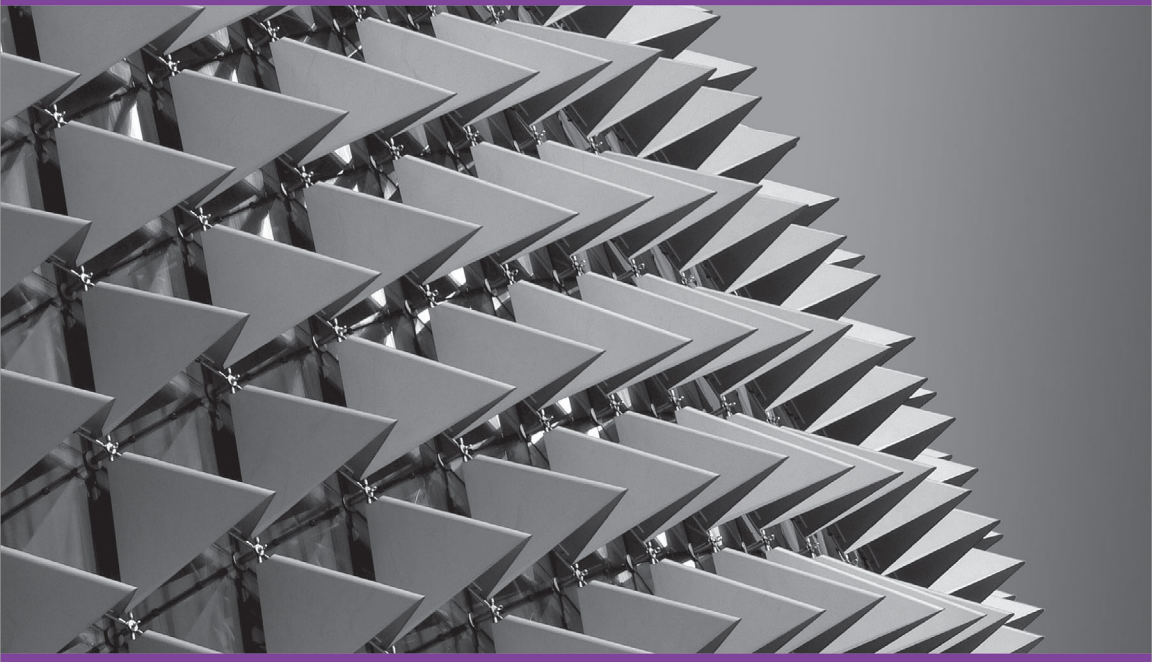
O'REILLY®

What is Serverless?

Understanding the Latest Advances in Cloud and Service-Based Architecture



Mike Roberts
& John Chapin

What is Serverless?

Understanding the Latest Advances in Cloud and Service-Based Architecture

Mike Roberts and John Chapin

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

What Is Serverless?
by Michael Roberts and John Chapin

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Foster
Production Editor: Colleen Cole
Copyeditor: Sonia Saruba
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

May 2017: First Edition

Revision History for the First Edition
2017-5-24: First Release

978-1-491-98416-1

[LSI]

Table of Contents

Preface

Fifteen years ago most companies were entirely responsible for the operations of their server-side applications, from custom engineered programs down to the configuration of network switches and firewalls, from management of highly available database servers down to the consideration of power requirements for their data center racks.

But then the cloud arrived. What started as a playground for hobbyists has become a greater than $10 billion annual revenue business for Amazon alone. The cloud has revolutionized how we think about operating applications. No longer do we concern ourselves with provisioning network gear or making a yearly capital plan of what servers we need to buy. Instead we rent virtual machines by the hour, we hand over database management to a team of folks whom we've never met, and we pay as much concern to how much electricity our systems require as to how to use a rotary telephone.

But one thing remains: we still think of our systems in terms of servers—discrete components that we allocate, provision, set up, deploy, initialize, monitor, manage, shut down, redeploy, and reinitialize. The problem is most of the time we don't actually care about any of those activities; all we (operationally) care about is that our software is performing the logic we intend it to, and that our data is safe and correct. Can the cloud help us here?

Yes it can, and in fact the cloud is turning our industry up on its head all over again. In late 2012, people started thinking about what it would mean to operate systems and not servers—to think of applications as workflow, distributed logic, and externally managed data stores. We describe this way of working as *Serverless*, not

because there aren't servers running anywhere, but *because we don't need to think about them anymore*.

This way of working first became realistic with mobile applications being built on top of hosted database platforms like Google Firebase. It then started gaining mindshare with server-side developers when Amazon launched AWS Lambda in 2014, and became viable for some HTTP-backed services when Amazon added API Gateway in 2015. By 2016 the hype machine was kicking in, but a Docker-like explosion of popularity failed to happen. Why? Because while from a management point of view Serverless is a natural progression of cloud economics and outsourcing, from an architectural point of view it requires new design patterns, new tooling, and new approaches to operational management.

In this report we explain what Serverless really means and what its significant benefits are. We also present its limitations, both inherent and implementation specific. We close with looking to the future of Serverless. The goal of this report is to answer the question, "Is Serverless the right choice for you and your team?"

CHAPTER 1

Introducing Serverless

In this chapter we're first going to go on a little history lesson to see what led us to Serverless. Given that context we'll describe what Serverless is. Finally we'll close out by summarizing why Serverless is both part of the natural growth of the cloud, and a jolt to how we approach application delivery.

Setting the Stage

To place a technology like Serverless in its proper context, we must first outline the steps along its evolutionary path.

The Birth of the Cloud

Let's travel back in time to 2006. No one has an iPhone yet, Ruby on Rails is a hot new programming environment, and Twitter is being launched. More germane to this report, however, is that many people are hosting their server-side applications on physical servers that they own and have racked in a data center.

In August of 2006 something happened which would fundamentally change this model. Amazon's new IT Division, Amazon Web Services (AWS), announced the launch of Elastic Compute Cloud (EC2).

EC2 was one of the first of many Infrastructure as a Service (IaaS) products. IaaS allows companies to rent compute capacity—that is, a host to run their internet-facing server applications—rather than buying their own machines. It also allows them to provision hosts

just in time, with the delay from requesting a machine to its availability being in the order of minutes.

EC2's five key advantages are:

Reduced labor cost
: Before Infrastructure as a Service, companies needed to hire specific technical operations staff who would work in data centers and manage their physical servers. This meant everything from power and networking, to racking and installing, to fixing physical problems with machines like bad RAM, to setting up the operating system (OS). With IaaS all of this goes away and instead becomes the responsibility of the IaaS service provider (AWS in the case of EC2).

Reduced risk
: When managing their own physical servers, companies are exposed to problems caused by unplanned incidents like failing hardware. This introduces downtime periods of highly volatile length since hardware problems are usually infrequent and can take a long time to fix. With IaaS, the customer, while still having some work to do in the event of a hardware failure, no longer needs know what to do to fix the hardware. Instead the customer can simply request a new machine instance, available within a few minutes, and re-install the application, limiting exposure to such issues.

Reduced infrastructure cost
: In many scenarios the cost of a connected EC2 instance is cheaper than running your own hardware when you take into account power, networking, etc. This is especially valid when you only want to run hosts for a few days or weeks, rather than many months or years at a stretch. Similarly, renting hosts by the hour rather than buying them outright allows different accounting: EC2 machines are an operating expense (Opex) rather than the capital expense (Capex) of physical machines, typically allowing much more favorable accounting flexibility.

Scaling
: Infrastructure costs drop significantly when considering the scaling benefits IaaS brings. With IaaS, companies have far more flexibility in scaling the numbers and types of servers they run. There is no longer a need to buy 10 high-end servers up front

because you think you might need them in a few months' time. Instead you can start with one or two low-powered, inexpensive instances, and then scale your number and types of instances up and down over time without any negative cost impact.

Lead time
: In the bad old days of self-hosted servers, it could take months to procure and provision a server for a new application. If you came up with an idea you wanted to try within a few weeks, then that was just too bad. With IaaS, lead time goes from months to minutes. This has ushered in the age of rapid product experimentation, as encouraged by the ideas in Lean Startup.

Infrastructural Outsourcing

Using IaaS is a technique we can define as *infrastructural outsourcing*. When we develop and operate software, we can break down the requirements of our work in two ways: those that are specific to our needs, and those that are the same for other teams and organizations working in similar ways. This second group of requirements we can define as *infrastructure*, and it ranges from physical commodities, such as the electric power to run our machines, right up to common application functions, like user authentication.

Infrastructural outsourcing can typically be provided by a service provider or vendor. For instance, electric power is provided by an electricity supplier, and networking is provided by an Internet Service Provider (ISP). A vendor is able to profitably provide such a service through two types of strategies: economic and technical, as we now describe.

Economy of Scale

Almost every form of infrastructural outsourcing is at least partly enabled by the idea of economy of scale—that doing the same thing many times in aggregate is cheaper than the sum of doing those things independently due to the efficiencies that can be exploited.

For instance, AWS can buy the same specification server for a lower price than a small company because AWS is buying servers by the thousand rather than individually. Similarly, hardware support cost per server is much lower for AWS than it is for a company that owns a handful of machines.

Technology Improvements

Infrastructural outsourcing also often comes about partly due to a technical innovation. In the case of EC2, that change was hardware virtualization.

Before IaaS appeared, a few IT vendors had started to allow companies to rent physical servers as hosts, typically by the month. While some companies used this service, the alternative of renting hosts by the hour was much more compelling. However, this was really only feasible once physical servers could be subdivided into many small, rapidly spun-up and down virtual machines (VMs). Once that was possible, IaaS was born.

Common Benefits

Infrastructural outsourcing typically echoes the five benefits of IaaS:

- Reduced labor cost—fewer people and less time required to perform infrastructure work
- Reduced risk—fewer subjects required to be expert in and more real time operational support capability
- Reduced resource cost—smaller cost for the same capability
- Increased flexibility of scaling—more resources and different types of similar resource can be accessed, and then disposed of, without significant penalty or waste
- Shorter lead time—reduced time-to-market from concept to production availability

Of course, infrastructural outsourcing also has its drawbacks and limitations, and we'll come to those later in this report.

The Cloud Grows

IaaS was one of the first key elements of the cloud, along with storage, e.g., the AWS Simple Storage Service (S3). AWS was an early mover and is still a leading cloud provider, but there are many other vendors from the large, like Microsoft and Google, to the not-yet-as-large, like DigitalOcean.

When we talk about "the cloud," we're usually referring to the *public cloud*, i.e., a collection of infrastructure services provided by a vendor, separate from your own company, and hosted in the vendor's

own data center. However, we've also seen a related growth of cloud products that companies can use in their own data centers using tools like Open Stack. Such self-hosted systems are often referred to as *private clouds*, and the act of using your own hardware and physical space is called *on-premise* (or just *on-prem*.)

The next evolution of the public cloud was Platform as a Service (PaaS). One of the most popular PaaS providers is Heroku. PaaS layers on top of IaaS, adding the operating system (OS) to the infrastructure being outsourced. With PaaS you deploy just applications, and the platform is responsible for OS installation, patch upgrades, system-level monitoring, service discovery, etc.

PaaS also has a popular self-hosted open source variant in Cloud Foundry. Since PaaS sits on top of an existing virtualization solution, you either host a "private PaaS" on-premise or on lower-level IaaS public cloud services. Using both public and private Cloud systems simultaneously is often referred to as *hybrid cloud*; being able to implement one PaaS across both environments can be a useful technique.

An alternative to using a PaaS on top of your virtual machines is to use containers. Docker has become incredibly popular over the last few years as a way to more clearly delineate an application's system requirements from the nitty-gritty of the operating system itself. There are cloud-based services to host and manage/orchestrate containers on a team's behalf, often referred to as Containers as a Service (CaaS). A public cloud example is Google's Container Engine. Some self-hosted CaaS's are Kubernetes and Mesos, which you can run privately or, like PaaS, on top of public IaaS services.

Both vendor-provided PaaS and CaaS are further forms of infrastructural outsourcing, just like IaaS. They mainly differ from IaaS by raising the level of abstraction further, allowing us to hand off more of our technology to others. As such, the benefits of PaaS and CaaS are the same as the five we listed earlier.

Slightly more specifically, we can group all three of these (IaaS, PaaS, CaaS) as *Compute as a Service*; in other words, different types of generic environments that we can run our own specialized software in. We'll use this term again soon.

Enter Serverless, Stage Right

So here we are, a little over a decade since the birth of the cloud. The main reason for this exposition is that Serverless, the subject of this report, is most simply described as the next evolution of cloud computing, and another form of infrastructural outsourcing. It has the same general five benefits that we've already seen, and is able to provide these through economy of scale and technological advances. But what is Serverless beyond that?

Defining Serverless

As soon as we get into any level of detail about Serverless, we hit the first confusing point: Serverless actually covers a range of techniques and technologies. We group these ideas into two areas: Backend as a Service (BaaS) and Functions as a Service (FaaS).

Backend as a Service

BaaS is all about replacing server side components that we code and/or manage ourselves with off-the-shelf services. It's closer in concept to Software as a Service (SaaS) than it is to things like virtual instances and containers. SaaS is typically about outsourcing business processes though—think HR or sales tools, or on the technical side, products like Github—whereas with BaaS, we're breaking up our applications into smaller pieces and implementing some of those pieces entirely with external products.

BaaS services are domain-generic remote components (i.e., not in-process libraries) that we can incorporate into our products, with an API being a typical integration paradigm.

BaaS has become especially popular with teams developing mobile apps or single-page web apps. Many such teams are able to rely significantly on third-party services to perform tasks that they would otherwise have needed to do themselves. Let's look at a couple of examples.

First up we have services like Google's Firebase (and before it was shut down, Parse). Firebase is a database product that is fully managed by a vendor (Google in this case) that can be used directly from a mobile or web application without the need for our own intermediary application server. This represents one aspect of BaaS: services that manage data components on our behalf.

BaaS services also allow us to rely on application logic that someone else has implemented. A good example here is authentication—many applications implement their own code to perform signup, login, password management, etc., but more often than not this code is very similar across many apps. Such repetition across teams and businesses is ripe for extraction into an external service, and that's precisely the aim of products like Auth0 and Amazon's Cognito. Both of these products allow mobile apps and web apps to have fully featured authentication and user management, but without a development team having to write or manage any of the code to implement those features.

Backend as a Service as a term became especially popular with the rise in mobile application development; in fact, the term is sometimes referred to as *Mobile Backend as a Service* (MBaaS). However, the key idea of using fully externally managed products as part of our application development is not unique to mobile development, or even front-end development in general. For instance, we might stop managing our own MySQL database server on EC2 machines, and instead use Amazon's RDS service, or we might replace our self-managed Kafka message bus installation with Kinesis. Other data infrastructure services include filesystems/object stores and data warehouses, while more logic-oriented examples include speech analysis as well as the authentication products we mentioned earlier, which can also be used from server-side components. Many of these services can be considered Serverless, but not all—we'll define what we think differentiates a Serverless service in Chapter 5.

Functions as a Service/Serverless Compute

The other half of Serverless is Functions as a Service (FaaS). FaaS is another form of Compute as a Service—a generic environment within which we can run our software, as described earlier. In fact some people (notably AWS) refer to FaaS as *Serverless Compute*. Lambda, from AWS, is the most widely adopted FaaS implementation currently available.

FaaS is a new way of building and deploying server-side software, oriented around deploying individual functions or operations. FaaS is where a lot of the buzz about Serverless comes from; in fact, many people think that Serverless *is* FaaS, but they're missing out on the complete picture.

When we traditionally deploy server-side software, we start with a host instance, typically a virtual machine (VM) instance or a container (see Figure 1-1). We then deploy our application within the host. If our host is a VM or a container, then our application is an operating system process. Usually our application contains of code for several different but related operations; for instance, a web service may allow both the retrieval and updating of resources.

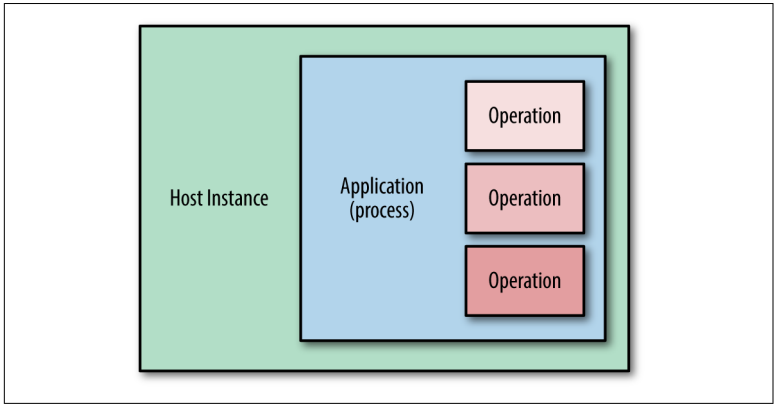


Figure 1-1. Traditional server-side software deployment

FaaS changes this model of deployment (see Figure 1-2). We strip away both the host instance and application process from our model. Instead we focus on just the individual operations or functions that express our application's logic. We upload those functions individually to a vendor-supplied FaaS platform.

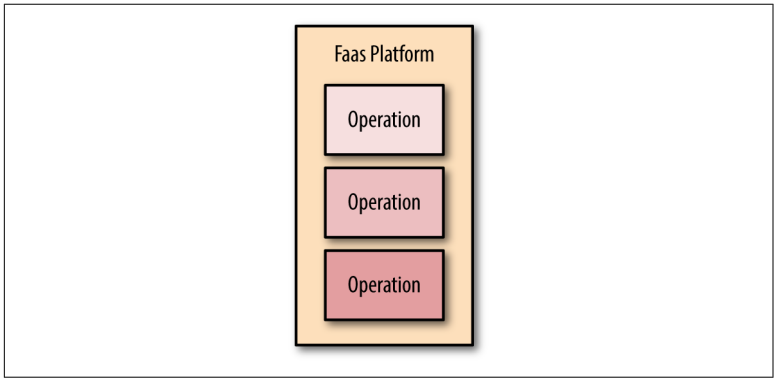


Figure 1-2. FaaS software deployment

The functions are not constantly active in a server process though, sitting idle until they need to be run as they would in a traditional system (Figure 1-3). Instead the FaaS platform is configured to listen for a specific event for each operation. When that event occurs, the vendor platform instantiates the Lambda function and then calls it with the triggering event.

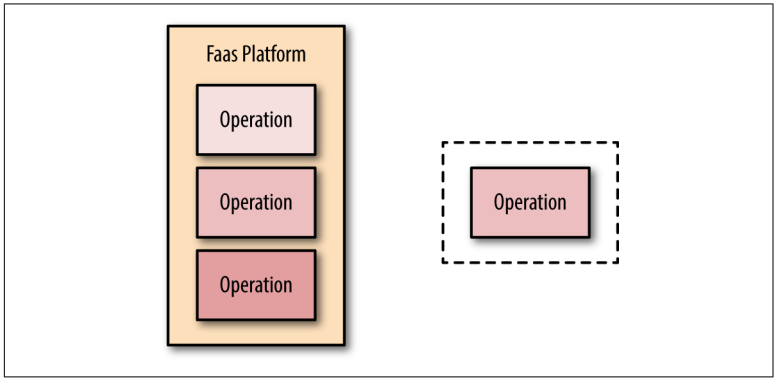


Figure 1-3. FaaS function lifecycle

Once the function has finished executing, the FaaS platform is free to tear it down. Alternatively, as an optimization, it may keep the function around for a little while until there's another event to be processed.

FaaS is inherently an event-driven approach. Beyond providing a platform to host and execute code, a FaaS vendor also integrates with various synchronous and asynchronous event sources. An example of a synchronous source is an HTTP API Gateway. An example of an asynchronous source is a hosted message bus, an object store, or a scheduled event similar to (cron).

AWS Lambda was launched in the Fall of 2014 and since then has grown in maturity and usage. While some usages of Lambda are very infrequent, just being executed a few times a day, some companies use Lambda to process billions of events per day. At the time of writing, Lambda is integrated with more than 15 different types of event sources, enabling it to be used for a wide variety of different applications.

Beyond AWS Lambda there are several other commercial FaaS offerings from Microsoft, IBM, Google, and smaller providers like Auth0. Just as with the various other Compute-as-a-Service plat-

forms we discussed earlier (IaaS, PaaS, CaaS), there are also open source projects that you can run on your own hardware or on a public cloud. This *private FaaS* space is busy at the moment, with no clear leader, and many of the options are fairly early in their development at time of writing. Examples are Galactic Fog, IronFunctions, Fission (which uses Kubernetes), as well as IBM's own OpenWhisk.

The Common Theme of Serverless

Superficially, BaaS and FaaS are quite different—the first is about entirely outsourcing individual elements of your application, and the second is a new hosting environment for running your own code. So why do we group them into the one area of Serverless?

The key is that *neither require you to manage your own server hosts or server processes*. With a fully Serverless app you are no longer thinking about any part of your architecture as a resource running on a host. All of your logic—whether you've coded it yourself, or whether you are integrated with a third party service—runs within a completely elastic operating environment. Your state is also stored in a similarly elastic form. *Serverless doesn't mean the servers have gone away, it means that you don't need to worry about them any more.*

Because of this key theme, BaaS and FaaS share some common benefits and limitations, which we look at in Chapters 3 and 4. There are other differentiators of a Serverless approach, also common to FaaS and BaaS, which we'll look at in Chapter 5.

An Evolution, with a Jolt

We mentioned in the preface that Serverless is an evolution. The reason for this is that over the last 10 years we've been moving more of what is common about our applications and environments to commodity services that we outsource. We see the same trend with Serverless—we're outsourcing host management, operating system management, resource allocation, scaling, and even entire components of application logic, and considering those things commodities. Economically and operationally there's a natural progression here.

However, there's a big change with Serverless when it comes to application architecture. Most cloud services, until now, have not

fundamentally changed how we design applications. For instance, when using a tool like Docker, we're putting a thinner "box" around our application, but it's still a box, and our logical architecture doesn't change significantly. When hosting our own MySQL instance in the cloud, we still need to think about how powerful a virtual machine we need to handle our load, and we still need to think about failover.

That changes with Serverless, and not gradually, but with a jolt. Serverless FaaS drives a very different type of application architecture through a fundamentally event-driven model, a much more granular form of deployment, and the need to persist state outside of our FaaS components (we'll see more of this later). Serverless BaaS frees us from writing entire logical components, but requires us to integrate our applications with the specific interface and model that a vendor provides.

So what does a Serverless application look like if it's so different? That's what we're going to explore next, in Chapter 2.

CHAPTER 2

What Do Serverless Applications Look Like?

Now that we're well grounded in what the term Serverless means, and we have an idea of what various Serverless components and services can do, how do we combine all of these things into a complete application? What does a Serverless application look like, especially in comparison to a non-Serverless application of comparable scope? These are the questions that we're going to tackle in this chapter.

A Reference Application

The application that we'll be using as a reference is a multiuser, turn-based game. It has the following high-level requirements:

- Mobile-friendly user interface
- User management and authentication
- Gameplay logic, leaderboards, past results

We've certainly overlooked some other features you might expect in a game, but the point of this exercise is not to actually build a game, but to compare a Serverless application architecture with a legacy, non-Serverless architecture.

Non-Serverless Architecture

Given those requirements, a non-Serverless architecture for our game might look something like Figure 2-1:

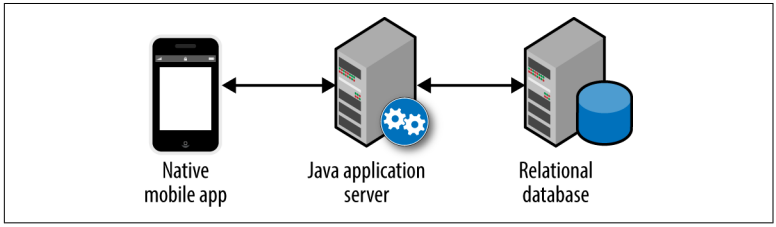


Figure 2-1. Non-Serverless game architecture

- A native mobile app for iOS or Android
- A backend written in Java, running in an application server, such as JBoss or Tomcat
- A relational database, such as MySQL

In this architecture, the mobile app is responsible for rendering a gameplay interface and handling input from the user, but it delegates most actual logic to the backend. From a code perspective, the mobile app is simple and lightweight. It uses HTTP to make requests to different API endpoints served by the backend Java application.

User management, authentication, and the various gameplay operations are encapsulated with the Java application code. The backend application also interacts with a single relational database in order to maintain state for in-progress games, and store results for completed games.

Why Change?

This simple architecture seems to meet our requirements, so why not stop there and call it good? Lurking beneath those bullet points are a host of development challenges and operational pitfalls.

In building our game, we'll need to have expertise in iOS and Java development, as well as expertise in configuring, deploying, and operating Java application servers. We'll also need to configure and operate the relational database server. Even after accounting for the application server and database, we need to configure and operate their respective host systems, regardless of whether those systems

are container-based or running directly on virtual or physical hardware. We also need to explicitly account for network connectivity between systems, and with our users out on the Internet, through routing policies, access control lists, and other mechanisms.

Even with that laundry list of concerns, we're still just dealing with those items necessary to simply make our game available. We haven't touched on security, scalability, or high availability, which are all critical aspects of a modern production system. The bottom line is that there is a lot of inherent complexity even in a simple architecture that addresses a short list of requirements. Building this system as architected is certainly possible, but all of that complexity will become friction when we're fixing bugs, adding features, or trying to rapidly prototype new ideas.

How to Change?

Now that we've uncovered some of the challenges of our legacy architecture, how might we change it? Let's look at how we can take our high-level requirements and use Serverless architectural patterns and components to address some of the challenges of the previous approach.

As we learned in Chapter 1, Serverless components can be grouped into two areas, Backend as a Service and Functions as a Service. Looking at the requirements for our game, some of those can be addressed by BaaS components, and some by FaaS components.

The Serverless Architecture

A Serverless architecture for our game might look something like Figure 2-2.

For example, while the user interface will remain a part of the native mobile app, user authentication and management can be handled by a BaaS service like AWS Cognito. Those services can be called directly from the mobile app to handle user-facing tasks like registration and authentication, and the same BaaS can be used by other backend components to retrieve user information.

With user management and authentication now handled by a BaaS, the logic previously handled by our backend Java application is simplified. We can use another component, AWS API Gateway, to handle routing HTTP requests between the mobile app and our

backend gameplay logic in a secure, scalable manner. Each distinct operation can then be encapsulated in a FaaS function.

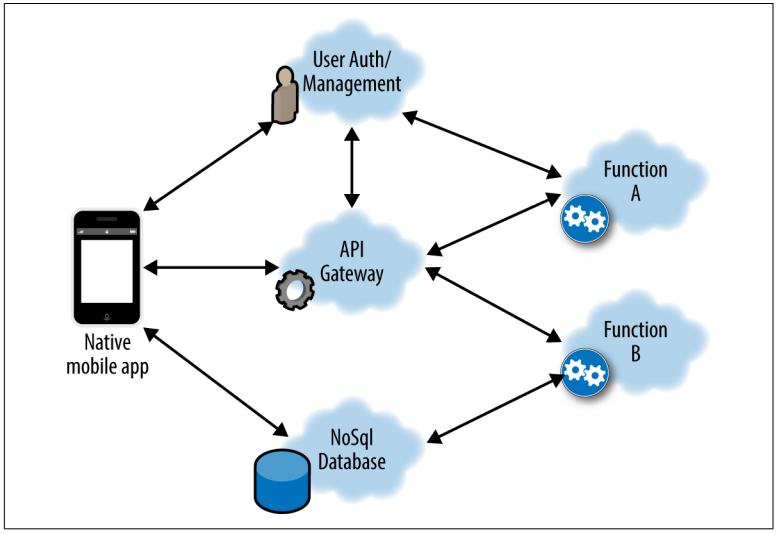


Figure 2-2. Serverless game architecture

What Is an API Gateway?

An API Gateway is a software component initially popular within the microservices world, but now also a key part of a HTTP-oriented Serverless Architecture. Amazon has its own implementation of an API Gateway named API Gateway.

An API Gateway's basic job is to be a web server that receives HTTP requests, routes the requests to a handler based on the route/path of the HTTP request, takes the response back from the handler, and finally returns the response to the original client. In the case of a Serverless architecture, the handler is typically a FaaS function, but can be any other backend service.

An API Gateway will typically do more than just this routing, also providing functionality for authentication and authorization, request/response mapping, user throttling, and more. API Gateways are configured, rather than coded, which is useful for speeding development, but care should be taken not to overuse some features that might be more easily tested and maintained in code.

Those backend FaaS functions can interact with a NoSQL, BaaS database like DynamoDB to manage gameplay state. In fact, one big change is that we no longer store any session state within our server-side application code, and instead persist all of it to the NoSQL store. While this may seem onerous, it actually significantly helps with scaling.

That same database can be seamlessly accessed by the mobile application to retrieve past results and leaderboard data. This allows us to move some business logic to the client rather than build it into the backend.

What Got Better?

This new Serverless architecture that we've described looks complicated, and it seems to have more distinct application components than our legacy architecture. However, due to our use of fully-managed Serverless components, we've removed many of the challenges around managing the infrastructure and underlying systems our application is using.

The code we write is now focused almost entirely on the unique logic of our game. What's more, our components are now decoupled and separate, and as such, we can switch them out or add new logic very quickly without the friction inherent in the non-Serverless architecture.

Scaling, high availability, and security are also qualities that are baked into the components we're using. This means that as our game grows in popularity, we don't need to worry about buying more powerful servers, wonder if our database is going to fall over, or troubleshoot a firewall configuration.

In short, we've reduced the labor cost of making the game, as well as the risk and resource costs of running it. All of its constituent components will scale flexibly. And if we have an idea for a new feature, our lead time is greatly reduced, so we can start to get feedback and iterate more quickly.

In Chapter 3 we're going to expand more on the benefits of Serverless, and in Chapter 4 we'll call out some of the limitations.

CHAPTER 3
Benefits of Serverless

In Chapter 1 we listed the five benefits of infrastructural outsourcing:

- Reduced labor cost
- Reduced risk
- Reduced resource cost
- Increased flexibility of scaling
- Shorter lead time

Serverless has elements of all five of these. The first four are all, to a greater or lesser extent, about cost savings, and this is what Serverless is best known for: *how to do the same thing you've done before, but cheaper*.

However, for us the cost savings are not the most exciting part of Serverless. What we get our biggest kick from is how much it reduces the time from conception to implementation, in other words, *how you do new things, faster*.

In this chapter we're going to dig into all these benefits and see how Serverless can help us.

Reduced Labor Cost

We said in Chapter 1 that Serverless was fundamentally about *no longer needing to look after your own server processes*—you care about your application's business logic and state, and you let someone else look after whatever else is necessary for those to work.

The first obvious benefit here is that there is less operations work. You're no longer managing operating systems, patch levels, database version upgrades, etc. If you're using a BaaS database, message bus, or object store, then congratulations—that's another piece of infrastructure you're not operating anymore.

With other BaaS services the labor benefits are even more clearly defined—*you have less logic to develop yourself*. We've already talked a couple of times about authentication services. The benefits to using one of these are that you have less code to define, develop, test, deploy, and operate, all of which takes engineering time and cost. Another example is a service like Mailgun which removes most of the hard work of processing the sending and receiving of email.

FaaS also has significant labor cost benefits over a traditional approach. Software development with FaaS is simplified because much of the infrastructural code is moved out to the platform. An example here is in the development of HTTP API Services—here all of the HTTP-level request and response processing is done for us by the API Gateway, as we described in Chapter 2.

Deployment with FaaS is easier because we're just uploading basic code units—zip files of source code in the case of Javascript or Python, and plain JAR files in the case of JVM-based languages. There are no Puppet, Chef, Ansible, or Docker configurations to manage. Other types of operational activity get more simple too, beyond just those we mentioned earlier in this section. For example, since we're no longer looking after an "always on" server process, we can limit our monitoring to more application-oriented metrics. These are statistics such as execution duration and customer-oriented metrics, rather than free disk space or CPU usage.

Not "NoOps"

There's been a fair amount of chatter in various places that all of this reduced operations work means that, in fact, we don't have any operations work left! After all, if we've gotten rid of bash scripting, Puppet/Chef configuration and OS-patch wrangling, what's left?

As you might have guessed from our tone, there's a lot left. Support, monitoring, deployment, security, and networking are still considerations when building a Serverless app, and while they may require less and/or different work, they do still need to be approached carefully, and with expertise. Serverless is not "NoOps."

For more on this we thoroughly recommend the work of Charity Majors. You can read some of what she has to say here and here.

Reduced Risk

When we think about risk and software applications we often consider how susceptible we are to failures and downtime. The larger the number of different types of systems, or components, our teams are responsible for managing, the larger the exposure to problems occurring. Instead of managing systems ourselves we can outsource them, as we've described previously in this report, and also outsource having to solve problems in those systems.

While overall we're still exposed to failure across all of the elements of the application, we've chosen to *manage* the risk differently—we are now relying on the expertise of others to solve some of those failures rather than fixing them ourselves. This is often a good idea since certain elements of a technical "stack" are ones that we might change rarely, and when failure does occur in them, the length of the downtime can be significant and indeterminant.

With Serverless we are significantly reducing the number of different technologies we are responsible for directly operating. Those that we do still manage ourselves are typically ones that our teams are working with frequently, and so we are much more able to handle failures with confidence when they occur.

A specific example here is managing a distributed NoSQL database. Once such a component is set up, it might be relatively rare that a

failure in a node occurs, but when it does, what happens? Does your team have the expertise to quickly and efficiently diagnose, fix, and recover from the problem? Maybe, but oftentimes not. Instead, a team can opt to use a Serverless NoSQL database service, such as Amazon DynamoDB. While outages in DynamoDB do occasionally happen, they are both relatively rare and managed effectively since Amazon has entire teams dedicated to this specific service.

As such, we say that risk is reduced when Serverless technologies are used since the expected downtime of components is reduced, and the time for them to be fixed is less volatile.

Reduced Resource Cost

Typically, when we are operating applications we have to figure out what kind of, and how many, underlying hosts they're going to run on. For example, how much RAM and CPU do our database servers need? How many different instances do we need to support scaling? Or to support high availability (HA)?

Once we've figured out *planning* what hosts or *resources* we need we can then work on *allocation*—mapping out which parts of our application are going to run on which resources. And finally, once we're ready to deploy our application, we need to actually obtain the hosts we wanted—this is *provisioning*.

This whole process is complicated, and it's far from an exact science. We very rarely know ahead of time *precisely* what our resource requirements are, and so we overestimate our plan. This is known as *over-provisioning*. This is actually the right thing to do—it's much better to have spare capacity and keep our application operating than for it to fall over under load. And for certain types of components, like databases, it may be hard to scale up later, so we might want to over-provision in anticipation of future load.

Over-provisioning means we're always paying for the capacity necessary to handle our peak expected load, even when our application isn't experiencing that load. The extreme case is when our application is sitting idle—at that point in time we're paying for our servers to be running when in fact they aren't doing anything useful. But even when our applications are active we don't want our hosts to be fully utilized. Instead, we want to leave some headroom in order to cope with unexpected spikes in load.

The huge benefit that Serverless brings to this area is that we don't plan, allocate, or provision resources. We let the service provide precisely the amount of capacity we need at any point in time. If we don't have any load, then we don't need any compute resource, and we won't pay for any. If we only have 1 GB of data, we don't need the capacity to store 100 GB. We trust that the service will scale up precisely when we need it, on demand, and this applies equally to FaaS and BaaS services.

Beyond the removal of the accounting headache that comes with resource allocation, Serverless also makes our costs far more efficient. For any application that has inconsistent load, we will see resource cost savings by using Serverless. For instance, if our application is only running for 5 minutes of every hour, we only pay for 5 minutes of every hour, not the whole 60 minutes. Further, a good Serverless product will have very precise increments of use; for example, AWS Lambda is charged by the 100 milliseconds of use, 36,000 times more precise than the hourly billing of EC2.

In modern non-Serverless apps, we do see some of these benefits through techniques like auto scaling; however, these approaches are often not nearly as precise as a Serverless product (see above our point of EC2 charging by the hour), and it's still typically not possible to auto-scale a non-Serverless database.

Optimization Drives Immediate Cost Savings

An interesting corollary to how we are charged for Serverless resources with this high level of precision is that when we optimize the performance of our application, through code or a change in architecture, we'll automatically see cost savings without having to change anything about our deployment.

For instance, say you have an application that processes HTTP requests and typically takes 500 ms to do so. Now let's imagine that we decide that's too slow, and so we optimize our system so that it now only takes 300 ms. Not only do we have happier users, we also just cut our compute costs by 40%—win, win!

Increased Flexibility of Scaling

All of these resource cost benefits come from the fact that a Serverless service precisely scales to our need. So how do we actually make that scaling happen? Do we need to setup auto-scale groups? Monitoring processes? No! In fact, scaling happens *automatically*, with no effort.

Let's take AWS Lambda as an example. When the platform receives the first event to trigger a function, it will spin up a container to run your code. If this event is still being processed when another event is received, the platform will spin up a second instance of your code to process the second event. This automatic, zero management, horizontal scaling will continue until Lambda has enough instances of your code to handle the load.

A particularly nice aspect to this is that Amazon will still only charge you for how long your code is executing, no matter how many containers it has to launch. For instance, it costs precisely the same to invoke a Lambda 100 separate times in one container sequentially as it does to invoke a Lambda 100 times concurrently in 100 different containers, assuming the total execution time across all the events is the same.

What Stops Lambda Scaling Infinitely?

What happens if someone performs a deliberate denial of service (DDoS) attack on your Lambda application—will it scale to tens of thousands of containers? No, fortunately not! Apart from anything else, this would get mighty expensive very quickly, and it would also negatively impact other users of the AWS platform.

Instead, Amazon places a concurrent execution limit on your account, in other words it will only spin up to a maximum number of Lambda containers across your whole AWS account. The default (at the time writing) for this is 1,000 concurrently executing Lambda functions, but you can request to have this increased depending on your needs.

Shorter Lead Time

The first four high-level benefits we've covered are all excellent reasons to consider Serverless—depending on what your application does you are likely to see significant (double digit percentage) cost savings by embracing Serverless technologies.

However, we'd like to show you a quote from Sree Kotay, the CTO of Comcast Cable, from an AWS Summit in August 2016. Full disclosure: he wasn't talking about Serverless, but he was talking about how Comcast had gained significantly from various other infrastructural outsourcing, moving from "on prem" to the cloud. He said the following:

> After going through this journey [of cloud and Agile] for the last five years we've realized benefits, and these benefits are around cost and scale. And they're critical and important, but interestingly they're not the compelling bit…*The key part is this really changes your innovation cycle*, it fundamentally shifts how you think about product development.
>
> —Sree Kotay

The point we want to make is that the CTO of a major corporation is saying that costs and scale aren't the most important thing to him —innovation is. So how does Serverless help in this regard?

Here are some more quotes, this time from Adrian Cockcroft (VP, Cloud Architecture Strategy at AWS, and formerly Cloud Architect at Netflix), talking about Serverless:

> We're starting to see applications be built in ridiculously short time periods.
>
> —Adrian Cockcroft

> Small teams of developers are building production-ready applications from scratch in just a few days. They are using short, simple functions and events to glue together robust API-driven data stores and services. The finished applications are already highly available and scalable, high utilization, low cost, and fast to deploy.
>
> —Adrian Cockcroft

Over the last few years we've seen great advances in improving the incremental cycle time of development through practices such as continuous delivery and automated testing, and technologies like

Docker. These techniques are great, but only once they are set up and stabilized. For innovation to *truly* flourish it's not enough to have short cycle time, you also need short *lead time*—the time from conceptualization of a new product or feature to having it deployed in a minimum viable way to a production environment.

Because Serverless removes so much of the incidental complexity of building, deploying, and operating applications in production, and at scale, it gives us a huge amount of leverage, so much so that our ways of delivering software can be turned upside down. With the right organizational support, innovation, and "Lean Startup" style, experimentation can become the default way of working for all businesses, and not just something reserved for startups or "hack days."

This is not just a theory. Beyond Adrian's quotes above, we've seen comparatively inexperienced engineers take on projects that would normally have taken months and required significant help from more senior folks. Instead, using a Serverless approach, they were able to implement the project largely unaided in a couple of days.

And this is why we are so excited about Serverless: beyond all of the cost savings, it's a democratizing approach that untaps the abilities of our fellow engineers, letting them focus on making their customers awesome.

CHAPTER 4

Limitations of Serverless

So far we've talked about what Serverless is and how we got here, shown you what Serverless applications look like, and told you the many wonderful ways that Serverless will make your life better. So far it's been all smiles, but now we need to tell you some hard truths.

Serverless is a different way of building and operating systems, and just like with most alternatives, there are limitations as well as advantages. Add to that the fact that Serverless is still new—AWS Lambda is the most mature FaaS platform, and its first, very limited version was only launched in late 2014.

All of this innovation and novelty means some big caveats—not everything works brilliantly well, and even those parts that do we haven't yet figured out the best ways of using. Furthermore, there are some implicit tradeoffs of using such an approach, which we discuss first.

Inherent Limitations

Some of the limitations of Serverless just come with the territory—we're never going to completely get around them. These are *inherent limitations*. Over time we'll learn better how to work around these, or in some cases even to embrace them.

State

It may seem obvious, but in a Serverless application, the management of state can be somewhat tricky. Aside from the components that are explicitly designed to be data stores, most Serverless components are effectively stateless. While this chapter is specifically about limitations, it's worth mentioning that one benefit of that statelessness is that scaling those components simply becomes a matter of increasing concurrency, rather than giving each instance of a component (like an AWS Lambda function) more resources.

However, the limitations are certainly clear as well. State*less* components must, by definition, interact with other, stateful components to persist any information beyond their immediate lifespan. As we'll talk about in the very next section, that interaction with other components inevitably introduces latency, as well as some complexity.

What's more, state*ful* Serverless components may have very different ways of managing information between vendors. For example, a BaaS product like Firebase, from Google, has different data expiry mechanisms and policies than a similar product like DynamoDB, from AWS.

Also, while statelessness is the fundamental rule in many cases, oftentimes specific implementations, especially FaaS platforms, *do* preserve some state between function invocations. This is purely an optimization and cannot be relied upon as it depends heavily on the underlying implementation of the platform. Unfortunately, it can also confuse developers and muddy the operational picture of a system. One knock-on effect of this opportunistic state optimization is that of inconsistent performance, which we'll touch on later.

Latency

In a non-Serverless application, if latency between application components is a concern, those components can generally be reliably co-located (within the same rack, or on the same host instance), or can even be brought together in the same process. Also, communication channels between components can be optimized to reduce latency, using specialized network protocols and data formats.

Successful early adopters of Serverless, however, advocate having small, single-purpose FaaS functions, triggered by events from other

components or services. Much of the inter-component communication in these systems happens via HTTP APIs, which can be slower than other transports. Interaction with BaaS components also follows a similar flow. The more components communicating over unoptimized channels, the more latency will be inherent in a Serverless application.

Another impact on latency is that of "cold starts," which we'll address a little later in this chapter.

While Serverless platform providers are always improving the performance of their underlying infrastructure, the highly-distributed, loosely coupled nature of Serverless applications means that latency will always be a concern. For some classes of problems, a Serverless approach may not be viable based on this limitation alone.

Local Testing

The difficulty of local testing is one of the most jarring limitations of Serverless application architectures. In a non-Serverless world, developers often have local analogs of application components (like databases, or message queues) which can be integrated for testing in much the same way the application might be deployed in production. Serverless applications can, of course, rely on unit tests, but more realistic integration or end-to-end testing is significantly more difficult.

The difficulties in local testing of Serverless applications can be classified in two ways. Firstly, because much of the infrastructure is abstracted away inside the platform, it can be difficult to connect the application components in a realistic way, incorporating production-like error handling, logging, performance, and scaling characteristics. Secondly, Serverless applications are inherently distributed, and consist of many separate pieces, so simply managing the myriad functions and BaaS components is challenging, even locally.

Instead of trying to perform integration testing locally, we recommend doing so remotely. This makes use of the Serverless platform directly, although that too has limitations, as we'll describe in the next section.

Loss of Control

Many of the limitations of Serverless are related to the reality that the FaaS or BaaS platform itself is developed and operated by a third party.

In a non-Serverless application, the entirety of the software stack may be under our control. If we're using open source software, we can even download and alter components from the operating system boot loader to the application server. However, such breadth of control is a double-edged sword. By altering or customizing our software stack, we take on implicit responsibility for that stack and all of the attendant bug fixes, security patches, and integration. For some use cases or business models, this makes sense, but for most, ownership and control of the software stack distracts focus from the business logic.

Going Serverless inherently involves giving up full control of the software stack on which code runs. We'll describe how that manifests itself in the remainder of this section.

Loss of control: configuration

An obvious limitation of Serverless is a loss of absolute control over configuration. For example, in the AWS Lambda FaaS platform, there are a very limited number of configuration parameters available, and no control whatsoever over JVM or operating system runtime parameters.

BaaS platforms are no different in this respect. The platform provider may expose some configuration, but it's likely to be limited or abstracted from however the actual underlying software is configured.

Loss of control: performance

Coupled closely with loss of control over configuration is a similar loss of control over the performance of Serverless components. The performance issue can be further broken down into two major categories: performance of application code and performance of the underlying Serverless platform.

Serverless platforms hide the details of program execution, in part due to the multiple layers of virtualization and abstraction that allow the platform operators to efficiently utilize their physical hardware.

If you have access to the physical hardware, core operating system, and runtime, it is straightforward to optimize your application code for peak performance on that hardware and software foundation. If your code is running in a container, which is itself running on a virtual server (like an EC2 instance), it becomes much more difficult to predict or optimize how your code might perform.

In observations of benchmarking code running on the AWS Lambda platform, we see that identically configured Lambdas can have drastically different performance characteristics. Those characteristics can vary for the same Lambda over the course of minutes or hours, as the underlying platform alters scheduling priorities and resource allocations in response to demand.

Similarly, the performance of BaaS platforms can be inconsistent from one request to the next. In combination with the loss of control over configuration, that inconsistency can be frustrating to encounter, especially when there are few options for resolution outside of raising a support ticket with the platform provider.

Loss of control: issue resolution

Once that support ticket is opened, however, who has the capability to resolve it? Issue resolution is another area in which we cede control to a vendor.

In a fully controlled system, if a hardware component has a fault, or the operating system requires a security patch, the owner of the system can take action to resolve issues. This extends into any infrastructure that the owner of the system also controls. In the case of noncritical issues, the system owner might choose to delay downtime or a maintenance window to a convenient time, perhaps when there is less load on the system or when a backup system might be available.

In a Serverless world, the only issues we can resolve are those within our application code, or issues due to the configuration of Serverless components and services. All other classes of issues must be resolved by the platform owner—we may not even know when or if an issue has occurred. AWS is well known for a lack of visibility into most issues with their underlying platforms, even serious ones. The AWS status page displays a sea of green checkmarks—only a sharp eye will pick out the occasional italicized "i" next to a green checkmark. That innocuous looking "i" represents nearly every state from

"the service had a few sporadic errors" to "an earthquake destroyed a data center." While it seems like understatement, it is also a testament to the global scale and resilience of the AWS infrastructure in that the loss of a data center is not necessarily a catastrophic event.

Loss of control: security

The last major aspect of loss of control that we're going to cover is security. As with issue resolution, the only opportunity to affect the security of a Serverless application is through the mechanisms supplied by the platform provider. These mechanisms often take the form of platform-specific security features instead of operating system level controls. Unfortunately, but unsurprisingly, those platform-specific security features are not generally compatible or transferable between platforms.

Furthermore, platform security controls may not meet the security requirements of your application. For example, all AWS API Gateways can be reached from anywhere on the public internet; access is controlled solely via API keys rather than any transport-based access controls. However, many internal applications are locked down via network controls. If an application should only be accessible from certain IP addresses, then API Gateway cannot be used.

Implementation Limitations

In contrast to all of the previous *inherent limitations*, *implementation limitations* are those that are a fact of Serverless life for now, but which should see rapid improvement as the Serverless ecosystem improves and as the wider Serverless community gains experience in using these new technologies.

Cold Starts

As we alluded to earlier, Serverless platforms can have inconsistent and poorly documented performance characteristics.

One of the most common performance issues is referred to as a *cold start*. On the AWS Lambda platform, this refers to the instantiation of the container in which our code is run, as well as some initialization of our code. These slower cold starts occur when a Lambda function is invoked for the first time or after having its configuration altered, when a Lambda function scales out (to more instances

running concurrently), or when the function simply hasn't been invoked in a while.

Once a container is instantiated, it can handle events without undergoing that same instantiation and initialization process. These "warm" invocations of the Lambda function are much faster. On the AWS Lambda platform, regularly used containers stay warm for hours, so in many applications cold starts are infrequent. For an AWS Lambda function processing at least one event per second, more than 99.99% of events should be processed by a warm container.

The difference between the "cold" and "warm" performance of FaaS functions makes it difficult to consistently predict performance, but as platforms mature, we feel that these limitations will be minimized or addressed.

Tooling Limitations

Given the newness of Serverless technologies, it's no surprise that tooling around deployment, management, and development is still in a state of infancy. While there are some tools and patterns out there right now, it's hard to say which tools and patterns will ultimately emerge as future "best practices."

Tooling limitations: deployment tools

Serverless deployment tools interact with the underlying platform, usually via an API. Since Serverless applications are composed of many individual components, deploying an entire application atomically is generally not feasible. Because of that fundamental architectural difference, it can be challenging to orchestrate deployments of large-scale Serverless applications.

Tooling limitations: execution environments

One of the most well publicized limitations of Serverless is the constrained execution environment of FaaS platforms. FaaS functions execute with limited CPU, memory, disk, and I/O resources, and unlike legacy server processes, cannot run indefinitely. For example, AWS Lambda functions can execute for a maximum of five minutes before being terminated by the platform, and are limited to a maximum of 1.5 GB of memory.

As the FaaS platform's underlying hardware gets more powerful, we can expect these resource limits to increase (as they already have in some cases). Further, designing a system to work comfortably within these limits often leads to a more scalable architecture.

Tooling limitations: monitoring & logging

One of the benefits of Serverless is that you're no longer responsible for many host- or process-level aspects of an application, and so monitoring metrics like disk space, CPU, and network I/O is not necessary (or, in fact, supported in many situations). However metrics more closely associated with actual business functionality still need to be monitored.

The extent to which monitoring is well supported in a Serverless environment is currently a mixed bag. As an example, AWS Lambda has a number of ways monitoring can be performed, but some of them are poorly documented, or at least poorly understood by most users. AWS also gives a default logging platform in CloudWatch Logs. CloudWatch Logs is somewhat limited as a log analysis platform (for example, searching over a number of different sources); however, it is fairly easy to export logs from CloudWatch to another system.

An area that is significantly lacking in support at present is distributed monitoring—that is the ability to understand what is happening for a business request as it is processed by a number of components. This kind of monitoring is under active development generally since it's also a concern for users of Microservices architectures, however Serverless systems will be much more easily operated once this kind of functionality is common place.

Tooling limitations: remote testing

In stark contrast to the inherent limitations of testing Serverless applications locally, the difficulty of remote testing is merely an implementation limitation. Some Serverless platform providers do make some remote testing possible, but typically only at the component level (for example, an individual function), not at the Serverless application level.

It can be difficult to exhaustively test a complex Serverless application without setting up an entirely separate account with the platform provider, to ensure that testing does not impact production

resources, and to ensure that account-wide platform limits are not exceeded by testing.

Tooling limitations: debugging

Debugging Serverless applications is still quite difficult, although due to their often stateless nature, there is less to be gained in introspection and runtime debugging. However, for thorny problems, there is no replacement for a runtime debugger that allows introspection and line-by-line stepping.

At the time of this writing, there is no production-ready capability to remotely debug AWS Lambda functions. Microsoft Azure Functions written in C# *can* be remotely debugged from within the Visual Studio development environment, but this capability doesn't exist for the other Azure Function language runtimes.

In addition to the limitations of debugging Serverless compute components, debugging Serverless applications as a whole is difficult, as it is with any distributed application. Services like AWS X-Ray are starting to enable distributed tracing of messages across Serverless infrastructure and components, but those tools are in their infancy. Third-party solutions do exist, but come with their own set of concerns and caveats, including integration challenges, performance impact, and cost. Of course, given the initial steps in this area, we can anticipate more progress in the near future.

Vendor Lock-In

Vendor lock-in seems like an obviously inherent limitation of Serverless applications. However, different Serverless platform vendors enforce different levels of lock-in, through their choice of integration patterns, APIs, and documentation. Application developers can also limit their use of vendor-specific features, admittedly with varying degrees of success depending on the platform.

For example, AWS services, while mostly closed-source and fully managed, are well documented, and at a high level can be thought of in abstract terms. DynamoDB can be thought of as simply a high-performance key-value store. SQS is simply a message queue, and Kinesis is an ordered log. Now, there are many specifics around the implementation of those services which make them AWS-specific, but as high-level components within a larger architecture, they

could be switched out for other, similar components from other vendors.

That being said, we of course must also acknowledge that much of the value of using a single Serverless vendor is that the components are well integrated, so to some extent the vendor lock-in is not necessarily in the components themselves, but in how they can be tied together easily, performantly, and securely.

On the other side of the vendor spectrum from AWS are platforms like Apache OpenWhisk, which is completely open source and not ostensibly tied to any single vendor (although much of its development is done by IBM to enable their fully-managed platform).

BaaS components, though, are somewhat more of a mixed bag. For example, AWS's S3 service has a published API specification, and other vendors like Dreamhost provide object storage systems that are API-compatible with S3.

Immaturity of Services

Some types of Serverless services, especially FaaS, work better with a good ecosystem around them. We see that clearly with the various services that AWS has built, or extended, to work well with Lambda.

Some of these services are new and still need to have a few more revisions before they cover a lot of what we might want to throw at them. API Gateway, for example, has improved substantially in its first 18 months but still doesn't support certain features we might expect from a universal web server (e.g., web sockets), and some features it does have are difficult to work with.

Similarly, we see brand-new services (at time of writing) like AWS Step Functions. This is a product that's clearly trying to solve an architectural gap in the Serverless world, but is very early in its capabilities.

Conclusion

We've covered the inherent and implementation limitations of Serverless in a fairly exhaustive way. The inherent limitations, as we discussed, are simply the reality of developing and operating Serverless applications in general, and some of these limitations are related to the loss of control inherent in using a Serverless or cloud platform.

While there may be some standardization in how we interact with platforms from different vendors, we're still ceding a substantial amount of control to the provider. Also, in some cases we'll find workarounds or opportunities for standardization (for example, AWS' Serverless Application Model, aka SAM).

The implementation limitations are also significant, but for the most part we can look forward to these limitations being addressed by platform providers and the wider community. As we gain collective experience in building and running Serverless applications, we will see most of these implementation limitations fall to the wayside in favor of well-designed and well-considered solutions.

CHAPTER 5

Differentiating Serverless

Everyone loves a bandwagon, and sure enough all kinds of projects and technologies are now declaring themselves Serverless. While in some ways this doesn't matter—it's what each individual tool or pattern does for you that's truly important—it is worth putting some more precise definition around the term *Serverless* so that we can at least clarify our understanding of what we're getting, and so that we can more effectively judge various technologies to see which is best for any given context.

In this chapter we don our flame-proof suits and differentiate what what we think is and isn't Serverless. Given these distinctions we then look at a selection of technologies, asking whether they should be considered Serverless, to help you in your architectural assessments.

The Key Traits of Serverless

In Chapter 1 we said that the most significant common theme across Serverless was the fact that you no longer need to manage your own server hosts or server processes. Or, in other words, that Serverless doesn't mean the servers have gone away, it means that you don't need to worry about them anymore.

That's by far the biggest change with regard to many other ways of delivering software, but there are others. Below we list what we think are the key defining criteria of a Serverless technology, whether it be in the realm of Backend as a Service or Functions as a

Service. There's no "standardization committee" to back up these opinions, but in our experience they are all good areas to consider when choosing a technology.

A Serverless service:

- Does not require managing a long-lived host or application instance
- Self auto-scales and auto-provisions, dependent on load
- Has costs that are based on precise usage, up from and down to zero usage
- Has performance capabilities defined in terms other than host size/count
- Has implicit high availability

Let's drill into these a little more.

Does not require managing a long-lived host or application instance
: This is the heart of Serverless. Most other ways of operating server-side software require us to deploy, run, and monitor an instance of an application (whether programmed by us or others), and that application's lifetime spans more than one request. Serverless implies the opposite of this: there is no long-lived server process, or server host, that we need to manage. That's not to say those servers don't exist—they absolutely do—but they are not our concern or responsibility.

  With Serverless FaaS we are still concerned with deploying software that we've written, but our technical monitoring should be on a per-request basis, or based on aggregate metrics across a number of requests.

Self auto-scales and auto-provisions, dependent on load
: Auto-scaling is the ability of a system to adjust capacity requirements dynamically based upon load. Most existing auto-scaling solutions require some amount of work by the utilizing team. Serverless *self* auto-scales from the first time you use it with no effort at all.

  Serverless is also *auto-provisioning* when it performs auto-scaling. It *removes all the effort of allocating capacity*, both in terms of *number* and *size* of underlying resources. This is a huge operational burden lifted.

Has costs that are based on precise usage, up from and down to zero usage

This is closely tied to the previous point, but Serverless costs are precisely correlated with usage. As we said in Chapter 3, if our application is only running for five minutes of every hour, we only pay for five minutes of every hour.

Similarly, the cost of using a BaaS database should be closely tied to usage, not capacity. This cost should be largely derived from actual amount of storage used and/or requests made.

Note that we're not saying costs should be *solely* based on usage—there will likely be some overhead cost for using the service in general (for instance, AWS charges a small amount for the artifacts you deploy to their Lambda platform)—but the lion's share of the costs should be proportional to fine-grained usage.

Has performance capabilities defined in terms other than host size/count

It's reasonable and useful for a Serverless platform to expose some performance configuration. For example, for AWS Lambda we can specify how much RAM we would like our environment to have (which also proportionally scales CPU and I/O performance). However, this configuration should be completely abstracted from whatever underlying instance or host types are being used.

For instance, when configuring an AWS Lambda function, we're never faced with decisions about EC2 instance types or sizes. We completely outsource those decisions to AWS. It is up to them whether they co-locate 100 Lambda functions on a powerful machine, or ten on a weaker one.

Has implicit high availability

When operating applications, we typically use the term *high availability* (HA) to mean that a service will continue to process requests even when an underlying component fails. With a Serverless service we typically expect the vendor to provide HA transparently for us.

As an example, if we're using a BaaS database we assume that the provider is doing whatever is necessary to handle the failure of individual hosts or internal components. Similarly, if we're using FaaS we expect that the FaaS provider will reroute

requests to a new instantiation of our function if the underlying host of the original instance becomes unavailable.

However, we may need to handle any upstream effects of an HA failover. For example, if an AWS Lambda function were to fail a few times as its host was becoming unstable, we need to handle those errors and retry at a later time.

Furthermore, we would not say that Serverless has implicit disaster recovery capability. Currently, AWS Lambda functions are deployed to a particular AWS "region." If that region were to fail, it is your responsibility to redeploy your application to a different region.

Is It Serverless?

Given the above criteria of Serverless, we can now consider whether a whole raft of technologies and architectural styles are, or are not, Serverless. Again, we are absolutely not saying that if a technology isn't Serverless it should be discounted for your particular problem. What we are saying is that if a technology is Serverless, you should expect it to have the previous list of qualities.

Unambiguously Serverless Technologies

We've already mentioned a number of technologies in this report that are unambiguously Serverless:

- Vendor-provided FaaS platforms, including AWS Lambda, Microsoft Azure Functions, etc.
- FaaS enabling services, like AWS API Gateway
- Authentication services, like Auth0 and AWS Cognito
- Mobile-app targeted databases, like Google Firebase

Less Obvious Serverless Technologies

There are some other products and services that we can describe as Serverless, even if in some cases they've been around longer than the term has been in use: AWS' Simple Storage Service (S3), and their messaging products Simple Queue Service (SQS) and Simple Notification Service (SNS), are all Serverless. All of them satisfy all of our criteria above.

We're also starting to see a new breed of AI-related Serverless technologies which satisfy our criteria. An example is Lex, Amazon's speech recognition product. It's not new for these kind of products to be offered as APIs, but the scaling and cost management properties of a Serverless BaaS are a new development.

Substantially Serverless Technologies

DynamoDB is Amazon's fully managed, NoSQL database product. Scaling of Dynamo is very clever—you don't need to consider instance types or sizes, you simply specify capacity in terms of provisioned throughput. However, this mechanism doesn't completely satisfy two of our Serverless criteria:

- By default the capacity for a Dynamo table, and the cost, don't scale automatically with load.
- The cost is never zero—even a minimally provisioned table incurs a small monthly cost.

We're going to give it the benefit of the doubt, though, because (1) capacity scaling can be automated using third-party tools, and (2) while costs can't quite reach zero, a small, minimally provisioned table costs less than a dollar per month.

Kinesis is another messaging product from Amazon, similar to Apache's Kafka. Like DynamoDB, capacity doesn't scale automatically with load, and costs never reach zero. However, also like DynamoDB, scaling can be automated, and the cost of a basic Kinesis stream is about 10 dollars per month.

The Worthy Mentions, but Not Quite Serverless

There are also many great services in this world that are fully managed, but don't quite fit our criteria. While AWS products like Relational Database Service (RDS) and Elasticache (their hosted version of Redis or Memcached) don't require any specific OS-level administration, they do not automatically scale, and are provisioned in terms of instances.

Serverless/Non-Serverless Hybrid Architectures

Sometimes it's not possible to build a purely Serverless system. Perhaps we need to integrate with existing non-Serverless endpoints. Or maybe there's an element of our architecture for which no current Serverless product is sufficient, possibly due to performance or security requirements.

When this happens, it's perfectly reasonable to build a hybrid architecture of both Serverless and non-Serverless components. For instance, using AWS, you may want to call an RDS database from a Lambda function, or invoke Lambda functions from triggers in RDS databases that use Amazon's home-grown Aurora engine.

When building a hybrid architecture, it's important to identify which elements are Serverless and which aren't in order to manage the effects of scaling. If you have a non-Serverless element downstream of a Serverless element, there is a chance the downstream element may be overwhelmed if the upstream one scales out wider than expected. As an example, this can happen if you call a relational database from a Lambda function.

When you have a situation like this, it is wise to wrap the non-Serverless element with another component that can throttle the request flow—perhaps a message bus or a custom service that you deploy traditionally.

Is PaaS Serverless?

Platform as a Service (PaaS) has many overlapping features and benefits with FaaS. It abstracts the underlying infrastructure and lets you focus on your application, simplifying deployment and operations concerns considerably. So is FaaS just another term for PaaS?

Adrian Cockroft, whom we also quoted in Chapter 3, tweeted this in 2016:

> "If your PaaS can efficiently start instances in 20 ms that run for half a second, then call it serverless."
>
> —Adrian Cockcroft

Most PaaS products cannot quickly instantiate applications and then tear them down. Most of them don't self auto-scale, and don't charge

purely based on precise usage, i.e., only when our code is processing an event.

Because of this we would argue that most PaaS products are not Serverless. There's an argument that Serverless FaaS is just a specific type of PaaS, but that's all getting a little convoluted to us!

Is CaaS Serverless?

FaaS is not the only recent trend to push on the idea of abstracting the underlying host that you run your software on. Docker exploded into our world only a few years ago and has been extremely popular. More recently, Kubernetes has taken up the challenge of how to deploy and orchestrate entire applications, and suites of applications, using Docker, without the user having to think about many deployment concerns. And finally, Google Container Engine provides a compelling cloud-hosted container environment (Containers as a Service, or CaaS), using Kubernetes.

But is CaaS Serverless?

The simple answer is no. Containers, while providing an extremely lean operating environment, are still based on an idea of running long-lived applications and server processes. Serverless properties like self auto-scaling (to zero) and self auto-provisioning are also generally not present in CaaS platforms. CaaS is starting to catch up though here, when we look at tools like the Cluster Autoscaler in Google Compute Engine.

We mentioned earlier in this report that there are also a few FaaS projects being built on top of Kubernetes. As such, even though CaaS itself isn't Serverless, the entire Kubernetes platform might offer a very interesting hybrid environment in a year or two.

CHAPTER 6
Looking to the Future

To close out this report we're going to gaze into our crystal ball and imagine what changes may happen with Serverless, and the organizations that use it, over the coming months and years.

Predictions

It's apparent that Serverless tools and platforms will mature significantly. We're still on the "bleeding edge" of many of these technologies. Deployment and configuration will become far easier, we'll have great monitoring tools for understanding what is happening across components, and the platforms will provide greater flexibility.

As an industry, we'll also collectively learn how to use these technologies. Ask five different teams today how to build and operate a moderately complex Serverless application, and you'll get five different answers. As we gain experience, we'll see some ideas and patterns form about what "good practice" looks like in common situations.

Our final prediction is that companies will change how they work in order to make the most of Serverless. As we said in Chapter 3, "*With the right organizational support,* innovation…can become the default way of working for all businesses." Those first five words are key—engineering teams need autonomy in order to exploit the lead time and experimentation advantages of Serverless. Teams gain this autonomy by having true product ownership and an ability to develop, deploy, and iterate on new applications without having to

wait on process roadblocks. Organizations that can manage themselves like this, while still maintaining budgetary and data safety, will find that their engineers are able to grow and be more effective by forming a deep understanding of what makes their customers awesome.

We give a more in-depth analysis of future trends in Serverless in our article, "The Future of Serverless Compute".

Conclusion

The goal of this report was to answer the question,"Is Serverless the right choice for you and your team?" For those of you who have, or can, embrace the public cloud; who have a desire to embrace experimentation as the way to produce the best product; and who are willing to roll up your sleeves and deal with a little lack of polish in your tools, we hope you'll agree with us that the answer is "yes."

Your next step is to jump in! Evaluating Serverless technology doesn't require an enormous investment of money or time, and most providers have a generous free tier of services. We provide a wealth of information and resources on our Symphonia website and blog, and we'd be excited to hear from you!

About the Authors

Mike Roberts is an engineering leader who has called New York City home since 2006. During his career he's been an engineer, a CTO, and other fun places in-between. Mike is a long-time proponent of Agile and DevOps values and is passionate about the role that cloud technologies have played in enabling such values for many high-functioning software teams. He sees Serverless as the next evolution of cloud systems and as such is excited about its ability to help teams, and their customers, be awesome.

John Chapin has over 15 years of experience as a technical executive and senior engineer. He was previously VP of Engineering, Core Services & Data Science at Intent Media, where he helped teams transform how they delivered business value through Serverless technology and Agile practices. Outside of Symphonia, he can be found running along the west side of Manhattan, surfing at Rockaway Beach, or planning his next trip abroad.

Mike and **John** are cofounders of Symphonia, an expert Serverless and cloud technology consultancy based in New York City.