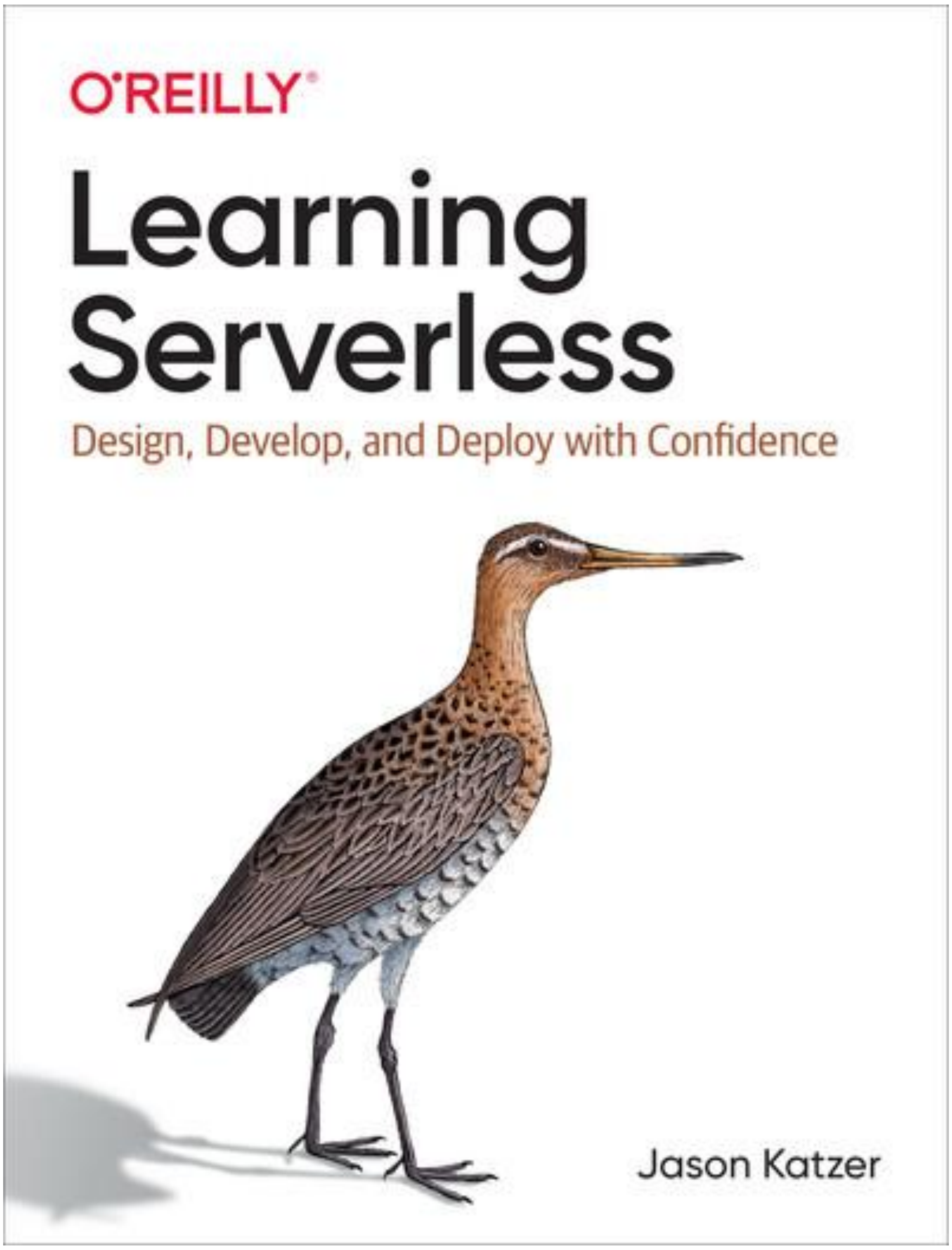

O'REILLY®

Learning Serverless

Design, Develop, and Deploy with Confidence

Jason Katzer

Introduction to Serverless

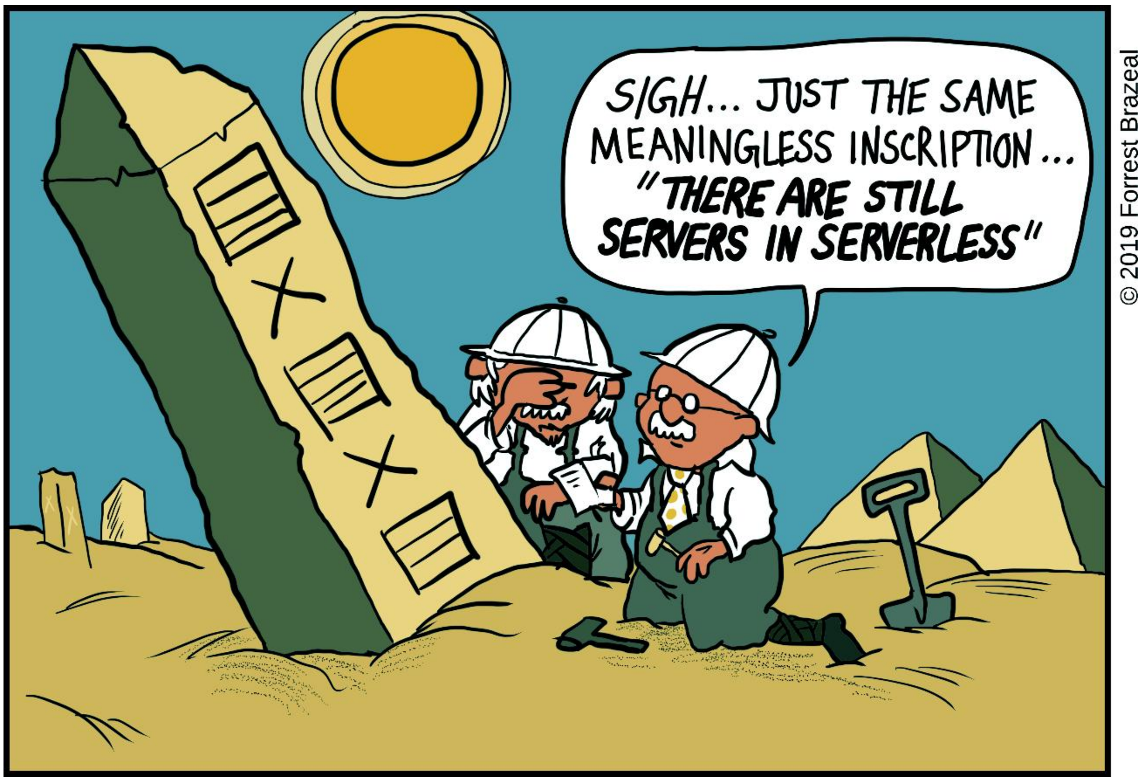
SIGH... JUST THE SAME MEANINGLESS INSCRIPTION... "THERE ARE STILL SERVERS IN SERVERLESS"
© 2019 Forrest Brazeal

Figure I-1. "Ancient Wisdom", from the webcomic FaaS and Furious by Forrest Brazeal, 2019

To begin...To begin...How to start? I'm hungry. I should get coffee. Coffee would help me think. Maybe I should write something first, then reward myself with coffee. Coffee and a muffin. Okay, so I need to establish the themes. Maybe a banana-nut. That's a good muffin.

Charlie Kaufman, *Adaptation*

What Is Serverless?

Serverless is the idea that you can run a server-based application without having to manage a server. If you have this text in front of you, there is a chance you are already familiar with serverless. But how do you explain it to others? Do you focus on what it is, a new way to run application code, or on what it isn't, *managing servers*? Do you next tell people about all of its weaknesses or its strengths? If your path to success with serverless involves others, and it likely does, you might be worried about how to best sell its benefits without scaring anyone away in the process. You see, serverless is still in its early days and it's on a path of continuous improvement. Some of its weaknesses are here to stay, but they are trade-offs to implementing the benefits and features that you are sure to love.

As the serverless community benefits from the rapid improvements of a cutting-edge technology, it can be a struggle to invent and adopt best practices. I hope not only to instill you with the most relevant best practices in serverless at the time of this writing, but to also help you create and adopt best practices. I want you to know enough of the rules to break them safely.

The first offering from Amazon Web Service (AWS) was the *Simple Storage Service* (S3). S3 allows you to store as many files as you like without having to provision any infrastructure. It is serverless storage. You may remember how S3 simplified storing arbitrary files: create a bucket (the S3 abstraction for a collection of files), then just give each *file* a unique name, and that's it. You don't have to worry about provisioning drives or backups, or the many other previous issues with storing files. Sure, S3 may not serve all file-storage purposes; it does have a maximum file size of 5 GB. Or it did until 2010, when this maximum was updated to 5 TB. How did they do it? AWS does all the heavy lifting on your behalf, splitting up files that are above 5 GB into multiple chunks in a way that is fully seamless to the user. That is the benefit of a serverless system. The similarities to modern serverless compute are uncanny: a general-purpose solution to a common problem (that may not fit all use cases), seamless improvements made behind the scenes (that usually began as hacks implemented by customers), and a pay-for-usage billing model.

The term *serverless* is a misnomer, because there are servers involved. In reality, there is no serverless, just someone else's container. The Cloud Native Computing Foundation's Serverless Working Group best summarizes this in a whitepaper:[1]

Serverless computing does not mean that we no longer use servers to host and run code; nor does it mean that operations engineers are no longer required. Rather, it refers to the idea that consumers of serverless computing no longer need to spend time and resources on server provisioning, maintenance, updates, scaling, and capacity planning. Instead, all of these tasks and capabilities are handled by a serverless platform and are completely abstracted away from the developers and IT/operations teams. As a result, developers focus on writing their applications' business logic.

Less time spent on wrangling infrastructure, and more time spent on shipping features. That is why serverless demand is increasing. There are limits to what it can do, but they are fading away as the technology progresses, and we may see functions become the new containers as cloud compute becomes increasingly managed. But how did we get here?

History of Serverless

Google's App Engine was one of the first popularized use examples of serverless. Launched in 2008, it was too early for the modern wave of serverless adoption. Many developers viewed App Engine as being too restrictive, and that it was more of a hobby offering from Google. In fact, despite being launched in 2008, it wasn't out of *preview* until 2011. But it was so ahead of its time that if you spin up a Google Cloud Function (at least in Python), it wraps your function in an App Engine-compatible package (using Flask) and runs it that way.

The term *serverless* first started swirling around in 2012, in an article written by Ken Fromm of Iron.io.[2] Fromm argued that web applications were moving from monolithic patterns to fully fledged distributed systems with loosely coupled components—which the next chapter will touch on. Fromm made his prediction more than two years before AWS released Lambda. But over six years before Lambda, and four years before this article, the first modern serverless system may have launched. Serverless offerings predate the term.

The Cloud Provider Landscape

You may not get to choose the cloud provider for the project you are working on. You may also be so picky that you only work at companies using certain cloud providers. Right now, we live in a time of great competition among the providers. They are constantly finding new dimensions to compete in, ranging from price, performance, capability, scalability, etc. However, this period will not last forever as business naturally transitions from high growth in an emerging market into a period where the ROI of developing new features, attracting clients based on price, and even support and engineering resources will no longer look attractive, and those things will get cut out as quickly as you can say "shareholder returns."

Think about the costs of storage, and network, especially egress. And sometimes that can be compounded when using external providers for things like logging or monitoring. Even transferring data between regions or availability zones of a given region may count the same as sending over the public internet, and may be caked into clicking a box such as "multi-AZ availability."

How easy is it to click a button and have a database that is available in data centers across the globe? Is that even something your organization would need or be allowed to use based on data protection laws? You aren't just renting a commodity offering. Even when it seems like you are, you are paying for a product. Evaluate it as such. Also consider the "cloud services" that may not fit into your mental model of what a cloud service is. For example, Google Meet is considered a Google Cloud product that is even marketed as being ready for telehealth. So is Google Maps. Amazon offers the ability to communicate with satellites, and to bring anywhere from a hardback book-sized device to an entire semi trailer to your site for easier migration of larger datasets to the cloud. Microsoft offers a lot of advanced functionality around its Office suite of products, which could be important for integrating with the software already in use by some in your organization.

Reliability, Availability, Disaster Recovery

What kind of SLAs/guarantees does the cloud provider offer? What is their track record? What are the remedies provided if they fail to meet their obligations? Google is known for services that stay in the beta phase too long, while AWS generally opts to offer a *preview* that may be more reliable but may still have some well-documented sharp edges.

Also consider how easy it is to build for reliability and availability on the foundation and services provided.

The network between the points of presence could be of interest if you plan on running a truly global service. This may be outside of your expertise and even your area of comfort, but some of these decisions may be made by taking a leap of faith in the right direction and realizing that any related issues will be understood. The best way to avoid these issues and to avoid surprises is with well-maintained documentation.

Amazon Web Services

Amazon Web Services (AWS) is oddly analogous to just being a data center with APIs. This is because of the services mandate at Amazon that states that all teams must build services, opening these very enterprise-y-feeling systems for public usage. This is the only explanation I can come up with that explains why it has so many sharp edges and weird quirks. It’s almost like joining Amazon as an engineer to build a greenfield project, and this is the internal service catalog. In fact, should you choose to use Amazon as your primary cloud provider, this mentality will help you make the most of the vendor lock-in to achieve maximum lift. AWS has the largest service catalog, although sometimes to its own detriment.

Google Cloud Platform

Google Cloud Platform (GCP) is a powerful cloud contender that is the closest approximation an outsider would have to Google’s own infrastructure. Kubernetes is a recreation based on its internal Borg platform, and follows its infrastructure as a service offering. Or that used to be the case. As the cloud wars heat up, Google has launched more competitive products that are marketing directly to the users of the public cloud, instead of relaunching its internal offerings once they have been in use for a number of years.

Microsoft Azure

This is a great choice if your organization is already all in on Microsoft. For example, if your organization uses Sharepoint, this would be the most straightforward way to trigger advanced workflows or custom logic in reaction to your company’s shared filesystem.

Tip

You don’t have to choose one of the big three to go serverless. If your organization is using Kubernetes, there are a number of open source options to run functions as you might run containers. Or even better, you can run containers as if they were functions. Knative, one of these options, is actually what powers Google Cloud Run. So don’t feel left out if your organization isn’t in the public cloud, or has gone “all-in” on Kubernetes. Running in Kubernetes may already come with its own sets of pros and cons that you may want to consider when going this route if you have other options.

Strengths of Serverless

Some of the strengths of serverless come from the change in focus from an application as a unit of deployment to a smaller and more finely grained model of individual functions. You can still choose to deploy an entire monolithic web application as one *function* and have it execute one API request per invocation, or you can choose to carve up your applications into individual functions to reap the most benefits of serverless. These are not the only two choices, as you can meet in the middle and use one function per service/microservice. But doing so is the same as utilizing containers: you won’t get all of the benefits of serverless, but you will still get the downsides. Meanwhile, some of the benefits of serverless are ones that you don’t want or need, and therefore become problems. Just as every coin has two sides, some of these benefits will directly map to a weakness.

Increased Scalability, Security, and Reliability

This is a core feature of the serverless experience. You don’t have to plan for future capacity, other than service limits from your cloud provider and interacting with nonserverless components or systems. For example, there was a big marketing campaign for new users on a project where I was using serverless. I found out the next day, which isn't ideal, but sure enough, Lambda and Amazon DynamoDB took on all the load without any action or knowledge from yours truly. You don't have to manage security other than the controls provided to you for granting permissions, and your application code and bundled libraries. When you have dedicated teams keeping up the servers that run your application code, you benefit from the economies of scale that provide maximum uptime.

You Only Pay for What You Use

One of the most attractive features of serverless compute is not paying for idle time. If your system is entirely serverless and isn't used in a given billing period, the total bill for compute will be $0. Pricing can be more predictable when you are charged for a specific number of tasks instead of instance hours. If you have an application that is used only during regular business hours and utilizes containers or other instances, you can automatically shut it down on the weekends to save money. But what happens if people need to use this service on the weekend? You leave it up and running in a minimal state, and wind up paying for every single weekend. What about holidays? What about a company all-hands event? You are paying for servers you don't need, but if you shut them off, your application has no availability. With serverless, a request automatically spins up the compute it needs if none is available, and you are only charged for that request. Your application is always available (although sometimes it may suffer from a cold start, which we will address later); if no one uses it, your cost for that time period is zero. Other parts of your application may have an effect on your cloud bill, such as data storage, monitoring, and other support systems, but the compute will be zero.

Saving Time and Money on Managing Servers

Of course, you'll be spending valuable engineering time on optimizing the cost of non-serverless systems. The time spent making those decisions isn't free. It is measured in the pay of engineers and the costs of recruiting and retaining them, as well as not shipping valuable features to users! Tedious tasks such as capacity planning don't entirely disappear when you use serverless, but you get to zoom out by an order of magnitude, and that has clear benefits.

Think of it this way: if you can't afford to hire a full-time platform-engineering team to run your code, why not rent one from your cloud provider? You may lose the ability to handle certain low-level tasks, but this is specialization of labor and economies of scale at their best. Instead of you having to manually configure autoscaling groups to provision and deprovision computing resources based on some abstractions of work that needs to be performed by your system, serverless specifically operates by scaling on the real metric of work that needs to be performed. There is no organization running in the cloud that does not have some amount of idle compute being wasted at any given time.

Improved Developer Productivity

Some cloud providers suggest using functions as glue to add logic and process to connect services. You don't have to reinvent the wheel when it comes to the distributed execution environment, queuing, retrying logic, and so on for modern serverless offerings that continue to increase with time.

There is no better example of this than creating an extract, transform, load (ETL) pipeline using serverless. An ETL pipeline takes data from one source, runs some compute over it, and loads it into a new destination. You can connect a data source that will automatically invoke a function for every single write performed on a database, and that lambda can transform that data without any servers or worrying about how many writes the original database will scale up or down to. It just works!

Decreased Management Responsibilities

I have already mentioned the idea of renting your DevOps from your cloud provider when your organization can't afford, or doesn't need a full-time dedicated team of platform engineers. That benefit cascades into other benefits as well. Serverless provides a stable container to target while having someone else manage security updates and patching of underlying infrastructure. It is important to remember the shared model of responsibility when utilizing any such offering, because you still have to take care of the security of your code and the libraries you utilize in your application. (I will cover security further in Chapter 9.) But you don't have to worry about patching the operating system, the libraries included on the system, and the version of the programming language itself. Your cloud provider employs a 24/7 staff of engineers who handle those choices and responsibilities.

Convenient Integrations

The biggest draw to the big three cloud providers when it comes to serverless is the integrations. It all comes down to the events. Publish a message in Google Cloud Pub/Sub? Why not react to that in real time with code? No need to monitor your worker nodes anymore. Add or update a record in your database, and boom, you can attach something that audits that action. Have a client upload an image directly to S3, and you can process that image into thumbnails without provisioning a single server. Using AWS Cognito to handle user accounts, and you want to send a welcome email after a user registration? Serverless handles all of those use cases and many more.

Current offerings provide a way to have your function code glue together actions in different parts of your system without worrying about provisioning queuing resources or creating a task execution and background work environment on your own. Some of this leads to opaqueness and comes back as a weakness in debugging. This is especially true as it becomes easier to glue together external services into your application architecture.

Weaknesses of Serverless

The most interesting part of the weaknesses of serverless is how they become less cumbersome or start to disappear as time progresses. The industry has seen major advances on some such issues while this book was being written, and change will continue to be rapid. To stay up to date on developments, especially in a space as rapidly evolving as serverless, or cloud native as a whole, make sure to follow the blog or get email announcements from your cloud providers, join mailing lists for relevant groups, or even follow developments on a site like Reddit.[3]

The Cold (Start) War

A *cold start* happens when a function invocation occurs and there is no running function available to execute the work. Instead, a new function container will spin up, and your users have to spend time waiting for your application to respond. Some people keep functions warm to prevent this problem, but I believe in using the right tool for the job.

People who are faking usage to keep their functions ready for user traffic are not using serverless as it was intended. What they really want is to instantly answer up to a certain number of concurrent requests without waiting for an additional machine to spin up and be added to a cluster. A serverless function will certainly beat spinning up an entire additional EC2 instance, but for some people that just isn't enough. I will give these users the benefit of the doubt by saying they are just so excited to use serverless that they are willing to use hacks to fix some of the weaknesses. If this form of latency is a deal breaker for your application, then serverless may not be right for your use case. Instead, utilize serverless for workloads that aren't directly user facing.

This cold start issue will continue to fade with time, but that future is already available now. Some environments offering compute at the edge or *Content Delivery Network* (CDN), such as Cloudflare workers, have increased limitations on the functions they will execute to decrease the cold start time in order to preprocess or post process a web request. Think about that. While most developers are trying to respond to API requests in under 100 ms, they are adding additional compute before or after that 100 ms. A common use case for this concept is injecting personalization into a cached page being served from a CDN.

Many companies offering are also alternative environments to solve this issue. It's an arms race. If you need the performance at this time, it may not be there. But it will get faster until it reaches the minimum overhead. AWS Lambda, for example, greatly improved its start-up time for cold starts by completely reinventing how it connects a function to a private network.

Compute Time

One agreed-upon weakness of serverless is the limited amount of time in which a particular workload can run. There are some workarounds, but it may make sense not to utilize serverless in some use cases.

However, this limitation is arbitrary in many ways. In 2018, Amazon changed the limits on Lambda from 5 minutes to 15 minutes. There was no need to rearchitect Lambda to make this change. As some issues with serverless are solved, the solutions will be available to you without any additional engineering overhead. You may have to spend engineering time to take the most advantage of the changing landscape, but your system will still work without those changes as well.

VPC/Network Issues

If your application needs to run inside a specific private subnet or cloud network, there are some limitations. You can't scale to 10,000 concurrent executions in a subnet with room for 254 IP addresses. Depending on your organization, you may be forced to operate in a virtual private cloud (VPC) in order to access private resources, or your application may call for accessing a database that can only be reached in a certain network. You will have to capacity plan to make sure your private networks are large enough. If you want to build a truly serverless system, you will have to avoid certain cloud offerings, persistence layers, or other design choices that will tie you to a specific private network.

Application Size

Limitations like compute time are also arbitrary, but if your application is too large, the cold start times may become unmanageable, so limiting the bundle size of your application is a good sanity check. How does this limitation affect you? One example is that you may not be able to ship a large Java application into a serverless function—using containers or instances is a better strategy for now, but keep an eye out for changes that could enable this. You may also be limited in the amount and size of dependencies of your application, although with the introduction of layers in AWS, there are advancements in this area as well.

Potential to Be More Expensive

If your application requires a predictable and stable amount of compute, you will overpay by using serverless. But consider the cost of maintenance and upkeep required for patching systems with security and other updates. You can pay your employees to do this, or you can *overpay* for your compute to have some of those maintenance costs bundled in. Does it make more sense to spend an extra $200,000 per year on a DevOps engineer or overpay on your cloud bill by $20,000 per year? Spend that money on another engineer who will build functionality with directly attributable revenue.

Vendor Lock-In

Every technology you select will likely lock you into using a specific vendor in one way or another: which base Docker image you use, which database you use, should you really add that additional package, and so on. You can lessen this by having your organization self-host a function execution environment on top of Kubernetes using open source software. But there is a high likelihood your organization already has some level of vendor lock-in to one of these cloud providers. If your organization has already made a trade-off in a specific direction, it makes sense to piggyback on top of that. This may be the case for you.

Vendor lock-in is an interesting concern. Some suggest this is just an overreaction to switching costs, which comes with all technology choices. They liken it to what happens if you want to change from Java to Python, or Go to Erlang. This is true only in that every developer has the choice of making optimizations and trade-offs as they see fit. Sure, you can save a lot of money on hosting costs by running your application on an old server under your desk, or on a cluster of Raspberry Pis, but you will likely choose to use virtualized instances from a large cloud provider because you will have to decide how you want to spend your time: writing code, or carrying buckets of diesel fuel up a staircase after a hurricane (see “The Physical World”).

Lock-in is something to be mindful of, but not to spend much time on. I will be focusing on examples primarily from AWS due to the depth of supporting services and integrations, but these examples are for illustration purposes. I am not advocating allegiance to any one particular provider, and think the most pragmatic approach is to keep your options open.

If your organization has chosen to invest in one of these platforms, take advantage of the deep service catalog you have available to get your job done in the best way with the fewest trade-offs possible. Learn to love your provider, but don’t trust them more than you should.

Complex Debugging

When you have a dynamic runtime, debugging can be complicated to reproduce errors in order to solve them. The more components or microservices your system is comprised of, the more difficult it can be to trace a user action throughout the entire system. That's why so many tools and SaaS offerings address these issues. I believe this is generally a symptom of using serverless incorrectly. Used correctly, serverless should give you more understanding of the core functionality of your systems. Some of these tools, however, are evolving into really compelling ways to find and filter issues, as well as providing data helpful in reproducing such errors. Your debugging and introspection are more powerful than ever before. What a time to be alive!

When Does It Make Sense to Use Serverless?

Many developers are making the move to serverless, or exploring serverless components for parts of their applications. Werner Vogels says:

At Amazon, we're not completely serverless ourselves, but we're moving in that direction. And so are many of our customers. In fact, we anticipate that there will soon be a whole generation of developers who have never touched a server and only write business logic. The reason is simple. Whether you're building net new applications or migrating legacy, using serverless primitives for compute, data, and integration enables you to benefit from the most agility that the cloud has to offer.

He sees serverless primitives (the most basic types of resources), as superior to their server-based equivalents, just as the cloud primitives were superior to the data center primitives were superior to the mainframe equivalents.

Use cases vary, but here are some of the most common and best reasons to use serverless.

The most important factor to determine your *success* will be the use case. Have you heard people complain about serverless? What do they talk about? Cold starts. While cold starts will eventually be optimized as close to zero as possible, you can build a system that is unaffected by cold starts. This pattern is the same for people who complain about how NoSQL doesn't have transactions, or how iPads don't have mouse support. Although these days, things are changing: DynamoDB offers NoSQL with transactions, and the latest iPad Pro has a trackpad.

You don't need a specific reason to use serverless, but here are some examples of the characteristics of the compute work you want to perform that will have the least friction and most benefit when utilizing serverless:

- Tasks that can be broken up into small independent units of work
- Tasks that either have infrequent or unpredictable usage patterns
- Background work, or system to system communication that will not be impacted by cold starts

Let's break these down. A *task* is a unit of work that isn't blocking, and can be broken up into smaller units of work that would each fit into a function.

Serverless is best used for load that is not predictable. This doesn't mean you can't use it in this case, it just may not be the most efficient and will cost more than use containers (but again, that doesn't include the overhead of managing the containers).

But what about your workload? If you can see your system as a collection of easily separable parts, and you don't want to deal with the overhead of servers for a lack of resources, it may make sense to use serverless.

Some parts of your application will be high velocity, at least when it comes to the rate of change of features and priorities. But then you have the strong and steady workhorse components. Imagine some of the problems you have yet to solve. There are some parts of your overall application that will be low velocity once version 1.0 is shipped. They don't directly serve users, but offload work from the application servers that do. Sending email to users is a perfect asynchronous task to set up to happen in the background that won't need much change to the basic architecture. It has somewhat unpredictable demand. And while you want it to happen in real time, the latency of a cold start is not going to ruin the password reset experience for a user locked out of their account.

Another interesting use case of serverless is the nearly infinite scale it brings. Let's assume it takes 30 seconds to process one minute of high resolution video for streaming. How long will it take to process a 90-minute film? 30 seconds. Because you can break up and parallelize the work and instantly feed it out to as many Lambda functions as possible, you can drastically speed up the time it takes to complete a task. This is actually one way Netflix uses serverless.

Another strong use case for serverless is event-driven architecture. Chapter 3 will cover serverless architectural patterns in detail.

One of the most helpful uses of serverless compute is that it acts as the glue between services. For example, you can monitor the utilization of a resource to scale a service up or down to save costs.

Want to resize uploaded images into thumbnails automatically without setting up a task or queueing service? Do you want to save money on your instances by using *spot* instances that cost less money than traditional instances? When those spot instances are being taken away (part of the reason they are less expensive), you can have a function automatically invoked on that cloud event to spin up a regular instance to take its place. Another spot instance becomes available later? Same thing in reverse: your function can spin up the instance that costs less money and terminate the more expensive one. Want to react to changes in data as they happen without adding brittle analytical code to the main conversion funnel of your application? Serverless can help with all of these use cases. It can be glue, DevOps, automation, out-of-band processing of data, or fully fledged applications.

When Is Serverless Compute Not Right for You?

Serverless will not serve you best when you have tasks that are computationally intensive, when your tasks have a long runtime that can't be broken up into smaller workloads, or when you need additional functionality not currently supported by the cloud providers, to name a few examples. These tasks might look like reading a large table of data and turning each row into an API request, encoding a feature-length film for streaming, or running a persistent WebSockets connection for a chat function. But some of these examples do have ways of being adapted to work. You can run a parallel scan or certain types of datastores such as DynamoDB. You can break up large files into smaller parallelized chucks as Netflix currently does to encode movies. You can use an API Gateway with WebSockets to maintain a real-time connection to clients, while invoking a Lambda for each message passed.

Let's Get Started

It is time to start or continue your serverless journey. By the end of this book you will have learned many fundamentals and best practices needed to succeed in any form of cloud computing, servers or not.

How "full stack" are you? If you specialize in certain areas, how did you choose those areas? Did you try other things out before deciding not to be an expert? You need to know how to manage servers before you can manage a system that manages them for you.

The choice to go serverless, is generally made to reduce the complexity in configuring and managing infrastructure, but you must have some basic understanding of the work you are abstracting away to build a reliable system on top of it. That will all be covered in this book.

Part I of this book will walk you through what it means to launch a proper production system. There will be servers involved, of course, but you won't need to know them personally. Part II will cover the tools you will need to be successful with serverless. Part III will cover some more advanced topics in depth, such as security.

Now let's talk about production systems.

1 Cloud Native Computing Foundation, "CNCF WG-Serverless Whitepaper v1.0," 2018, *https://oreil.ly/A2ehY*.

2 Ken Fromm, "Why the Future of Software and Apps Is Serverless," Read/Write, *https://oreil.ly/vh5ck*.

3 Personally, I like to stay up to date with Hacker News.