STACKERY

SERVERLESS DevOps

WHAT DO WE DO WHEN THE SERVER GOES AWAY?

SERVERLESSOPS

Tom McLaughlin

Table of Contents

Chapter 1:
Introduction

"WHAT DO WE DO WHEN THE SERVER GOES AWAY?"

When I built my first serverless application using AWS Lambda, I was excited right from the start. It gave me the opportunity to spend more time building my application and less time focusing on the infrastructure that was required to run it. I didn't have to think about how to get the service up and running, or evven ask permission for the necessary resources. The result was an application that was running and doing what I needed more quickly than I had ever experienced before.

But that experience also led me to ponder my future. If there were no servers to manage, what would I do? Would I be able to explain my job? Would I be able to explain to an employer (current or prospective) the value I provide? **This was why I started ServerlessOps.**

The Changing Landscape of Operations

I have seen, and personally been affected by, a shift in the operational needs of an organization due to changes in technology. I once sat in a meeting in which engineering leadership told me there would come a day when my skills would no longer be necessary. When that happened — and they assured me it would be soon — I would not be qualified to be on the engineering team any longer.

Now is the right time for us to begin discussing what operations will be in a serverless world. What happens if we don't? It will be defined for us.

At one end of the spectrum, there are **people proposing NoOps**, where all operational responsibilities are transferred to software engineers. That view exposes a fundamental misunderstanding of operations and its importance. Fortunately, larger voices are already out there **countering that attitude**.

At the other end, there are people who believe operations teams will always be necessary and the status quo will remain. That view simply ignores the change that has been occurring over the past several years.

If DevOps and public cloud adoption hasn't affected your job yet, it's only a matter of time. Adopting a they'll-always-need-me-as-I-am-today attitude leaves you unprepared for change.

Somewhere in between those views, an actual answer exists. Production operations, through its growth in complexity, is expanding and changing shape. As traditional problems we deal with today become abstracted away by serverless, we'll see engineering teams

and organizations change. (This will be particularly acute in SaaS product companies.) But many of today's problems — system architecture design, deployment, security, observability, and more — will still exist.

The serverless community **largely recognizes the value of operations** as a vital component of going serverless successfully. Operations never goes away; it simply evolves in practice and meaning. Operations engineers and their expertise still possess tremendous value. But, as a community, we will have to define a new role for ourselves.

Starting the Conversation

In this ebook, we will discuss:

- **Operational concerns and responsibilities when much of the stack has been abstracted away**
- **A proposed description of the role of operations when serverless**

This ebook is a start at defining what I see as the role for operations in a serverless environment. I don't believe, however, it's the only way to define the role. I think of operations in the context of SaaS startup companies.

It has been awhile since I worked on traditional internal IT projects or thought of engineering without a more product growth-oriented mindset. My problems and experiences aren't necessarily your problems and experiences. This is the start of a conversation.

Personal Biases

As you read this, keep a few things in mind. What I discuss on a technical level is very Amazon Web Services (AWS) centric. This is just a matter of my own experience and the cloud platform I'm most familiar with. You can apply these same ideas to serverless on Microsoft Azure or Google Cloud.

What I write, however, assumes public cloud provider serverless and not private platforms. The effects of public cloud serverless are more far reaching and disruptive than private cloud serverless.

In addition, I've worked primarily in product SaaS companies and startups for the past several years. My work has contributed toward the delivery of a company's primary revenue-generating service. But you can take many of these lessons and reapply them. Your customer doesn't need to be external to your organization. They can just as easily be your coworker.

With all that in mind, here's what I see as the future serverless operations.

Chapter 2:

What is Serverless?

"YES, SERVERLESS HAS SERVERS."

Before we can explain the impact of serverless on operations engineers, we need to be clear about what we're discussing. Serverless is a new concept and its meaning is still vague to many people. Even more confusing, people in the serverless world can disagree on what the word means. For that reason we're going to establish what we mean by serverless.

What Is Serverless?

To start, let's give a brief explanation of what serverless is. Serverless is a cloud systems architecture that involves no servers, virtual machines, or containers to provisionor manage. They still exist underneath the running application, but their presence is abstracted away from the developer or operator of the serverless application. Similarly, if you've adopted public cloud virtualization already, you know the underlying hardware is no longer your concern.

Serverless is often, incorrectly, reduced to Functions as a Service (FaaS). It's viewed as just another component of the Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Containers as a Service (CaaS) evolution. But it's more than that. You can manage to build serverless applications without a FaaS, e.g. AWS Lambda, component. For example, you can have a web application composed of HTML, CSS, graphics, and client-side JavaScript. Hosted with AWS CloudFront and S3, and it's a serverless application.

So what makes something serverless? What would make a simple web application serverless but an application inside of a Docker container not?

These four characteristics are used by AWS to classify what is serverless. They apply to serverless cloud services and applications as a whole. You can use these characteristics to reasonably distinguish what is and what is not serverless.

No servers to manage or provision: You're not managing physical servers, virtual machines, or containers. While they may exist, they're managed by the cloud provider and inaccessible to you.

Priced by consumption (not capacity): In the the serverless community you often hear, "You never pay for idle time." If no one is using your service, then you aren't paying for it.

With AWS Lambda you pay for the amount of time your function ran for, as opposed to an EC2 instance where you pay for the time the instance runs as well as the time it was idle.

Scales with usage: With non-serverless systems we're used to scaling services horizontally and vertically to meet demand. Typically, this work was done manually until cloud providers began offering auto-scaling services.

Serverless services and applications have auto-scaling built in. As requests come in, a service or application scales to meet the demand. With auto-scaling, however, you're responsible for figuring out how to integrate the new service instance with the existing running instances. Some services are easier to integrate than others. Serverless takes care of that work for you.

Availability and fault tolerance built in: You're not responsible for ensuring the availability and fault tolerance of the serverless offerings provided by your cloud provider. That's their job. That means you're not running multiple instances of a service to account for the possibility of failure. If you're running RabbitMQ, then you've set up multiple instances in case there's an issue with a host. If you're using AWS SQS, then you create a single queue. AWS provides an available and fault tolerant queuing service.

Public vs. Private Serverless

Increasingly, all organizations are becoming tech organizations in one form or another. If you're not a cloud-hosting provider, then cloud infrastructure is undifferentiated work; work a typical organization requires to function. One of the key advantages of serverless is the reduction in responsibilities for operating cloud infrastructure. It provides the opportunity to reallocate time and people to problems unique to the organization.

That means greater emphasis up the technical stack on the services that provide the most direct value in your organization. Serverless also allows for faster delivery of new services and features. By removing infrastructure as a potential roadblock, organizations can deliver with one less potential friction point.

There's both public cloud provider serverless options from **Amazon Web Services (AWS)**, **Microsoft Azure**, and **Google Cloud**; as well as private cloud serverless offerings like **Apache OpenWhisk** and **Google Knative**, which are both for **Kubernetes**. For the purposes of this piece, we're only considering public cloud serverless, and we use AWS examples.

We only consider public cloud serverless because, to start, private cloud serverless isn't particularly disruptive to ops. If your organization adopts serverless on top of Kubernetes, then the work of operations doesn't really change. You still need people to operate the serverless platform.

The second reason we only consider public cloud serverless is more philosophical. It goes back to the same reasons we largely don't consider on-prem "cloud" infrastructure in the same light as public cloud offerings. On-prem cloud offerings often negate the benefits of public cloud adoption.

The same is true of public versus private serverless platforms. Private serverless violates all four characteristics that make something serverless. You still have to manage servers, you pay regardless of platform use, you still need to plan for capacity, and you're responsible for its availability and fault tolerance.

More importantly, many of the benefits of serverless are erased. There's no reduction of undifferentiated work. No reallocation of people and time to focus further up the application stack. And infrastructure still remains a potential roadblock.

In the end, you're left with all the complexity of running and managing a serverless platform combined with the new complexity of serverless applications.

More Than Just Tech

Something important to realize most about serverless is that it is more than just something technical. In the debate between private and public serverless, the criticisms of private serverless are not technical. They are criticisms about the inability to fully realize the value of serverless as a technology.

As a historical analogy, look to what made public cloud adoption so successful, or even how it failed in some organizations. Public cloud adoption lead to us rethinking how we architected applications, how our teams worked together, and our expectations of what was possible or even acceptable in how engineers interacted with computing resources. Contrast those experiences with others who saw public cloud, or even private cloud, as just a new form of host virtualization. No technical, organizational, or expectation changes. How did those organizations fair in comparison? What made public cloud adoption so influential wasn't the technology, but how our organizations changed as a result of it.

While this ebook covers technical aspects of serverless, it also covers more importantly the impact it will have on operations and the changes brought with it. As you read this always be asking yourself, "How does this technology change things?"

Chapter 3:

Where Does Ops Belong?

"GET RID OF YOUR OPERATIONS TEAM."

Just as the term serverless can be confusing, so too can be operations. People picture different things when the term is used and this makes conversation confusing. I have a fairly expansive definition of what operations is. This often leads to conversations where I hear people propose a scenario where operations does not exist, and what they propose as a replacement is still what I consider to be operations. How can we discuss the impact of serverless on operations when we can't even agree on what we're talking about?

Once we have a common understanding of what we're talking about, let's then establish where operations people belong. I don't think operations teams make much sense in a serverless environment. But, operations people hold useful value. So where do they go?

What Is Operations?

People's understanding of operations is often different, but usually correct. The choice of definition is just a signal of a person's experiences, needs, and priorities; and ultimately their point of view. Those divergent definitions, however, often result in disjointed conversations where people talk past one another.

At the highest level, operations is the practice of keeping the tech that runs your business going. But if you dig a little deeper, the term operations can be used in a variety of ways. Because these meanings were tightly coupled for the longest time people tend to conflate them.

What is operations? It is a:

- **A team**
- **A role**
- **A responsibility**
- **A set of tasks**

Traditionally, the role was held by the operations engineer on the operations team who had operational responsibility and performed operations-related tasks. The introduction of DevOps in recent years has changed that set-up significantly. The rigid structure of silos that once kept all of these definitions with one person were torn down. And with that the definitions themselves broke apart.

On one side of the DevOps adoption spectrum, operations teams and the role of individuals remained largely unchanged. Both developers and operations remained separate teams but with some operations responsibility crossing over onto development teams while both operations and development experienced higher than previous levels of communication

between each other. We see this change in operational responsibility when someone says "developers get alerts first," in their organization. Or, when someone says developers are no longer able to toss code "over the wall".

On the other end of the adoption spectrum, operations teams and people went away. Some organizations grouped engineers with operational and software development skills together to create cross-functional teams. Those organizations don't have an ops team, and they don't plan on having one.

In the middle of these two ends of the adoption spectrum, the operations role varies. In some organizations, it has changed little beyond the addition of automation skills (e.g. Puppet, Ansible, and Chef). Other teams have seen automation as a means to an end for augmenting their operations responsibilities. And in some cases, operations engineers trend much closer toward a developer; a developer of configuration management and other tooling.

So what does serverless hold for these definitions of operations?

What Will Operations Be With Serverless?

Here is the future I see for operations for serverless infrastructure:

1. **The operations team will go away.**
2. **Operations engineers will be absorbed by development teams.**
3. **Operations engineers will be responsible for needs further up the application stack.**

Serverless operations is not NoOps. It is also not an anti-DevOps return to silos. If the traditional ops teams is dissolved, those engineers will require new homes. Their homes will be individual development teams, and those teams will be product pods or feature teams.

That will lead to a rapid rise in the formation of fully cross-functional teams who can handle both development and operations. Finally, many operations engineers will find themselves applying their strengths deeper into software applications than they may be used to.

Why Dissolve the Ops Team?

When we picture the pre-DevOps world, we see two silos: one developers and one operations, with a wall between them. Developers were often accused of tossing engineering over a wall and onto an unsuspecting operations organization. When DevOps came, we tore that wall down, and the two became more collaborative.

But what "tearing down the wall" was in practice varied by organization. In some cases it just meant more meetings. Now someone warned operations before they threw something at them.

But you still had capability-aligned, independent teams that were required to work together to deliver a solution. And in reality, there were probably more than two teams involved.

Who else was involved? There was a project or product manager who oversaw delivery and ensured it satisfied the needs of the organization. If you were in a product or SaaS company, there was perhaps UX and design ensuring the usability and look of the product.

There may have been more than one development team involved; frontend and backend development may be different teams. All of those independent, capability-aligned teams needed to work together to deliver a solution.

Product and SaaS companies in particular realized this entire process was inefficient. So organizations started realigning teams away from functional capabilities and toward product, feature, or problem domains. Those feature teams, or product pods, or whatever they were called were cross-functional teams aligned along solving specific problems.

What do those teams look like now? Where I've seen them in use they have typically resembled the following:

- **Product or project manager**
- **Engineering lead**
- **Frontend developer(s)**
- **Backend developer(s)**
- **Product designer and/or UX researcher (often the same person)**

The product or project manager (PM) is responsible for owning and representing the needs of the business. The PM's job is to turn the needs of the business into clear objectives and goals, while also leading the team effort in coming up with ideas to achieve success.

The product designer or UX researcher works with the PM to gather user data and turn ideas into designs and prototypes. The tech lead is responsible for leading the engineering effort by estimating the technical work involved and guiding frontend and backend engineers appropriately.

What you end up with is a single team with multiple capabilities all moving in the same direction who are a part of the process from start to finish. The team is made stronger by their cross-functional skill set, which leads to the delivery of better solutions and services.

Operations, however, was often left out of that realignment. (Though sometimes operations became their own cross-functional team delivering services to the other teams.) The needs of operating infrastructure were often too big for a single person on a team to handle. So while other parts of the organization realigned, operations remained as a separate capability-aligned team.

This has worked well enough for a long time. Infrastructure was not easy enough for many teams to reliably deliver and operate without detracting from their primary problem domain.

But serverless disrupts that relationship. It's now easy enough for a developer to deliver their own cloud infrastructure. In fact, they need to, since serverless combines both infrastructure configuration and application code together.

A development team doesn't need the operations team to deliver a solution. There's no way for a separate operations team to insert itself into the service-process without regressing to the role of gatekeeper. And that gatekeeper role has been going away in many organizations for years.

The need for operations teams to change is driven not by the technical effects of serverless but by how serverless will affect the way organizations and their people function and carry out their role.

Remaining as a capability-aligned operations team when your devs no longer need you for infrastructure means you'll largely become invisible. They'll also stop going to you for smaller problems as they encounter them and largely choose to solve those problems on their own.

Sooner or later, the team is no longer a part of the service delivery process. And eventually, someone will ask why the team exists at all. You're in a bad spot professionally when people are questioning why your job exists.

But operations, as in the responsibility and tasks, is still important. Those functions are still required for building serverless systems. The decreased usefulness and capability to perform by an operations team but the need for their skills means it's time to rethink where operations people belong. That's why it's time for operations teams to dissolve and its members to join product pods and feature teams.

The Ops Role in a Product Pod

What will be the role of the operations engineer as a product pod member be? Their high-level responsibility will be the health of the team's services and systems. That doesn't mean they're the first one paged every time. It means that they will be the domain expert in those areas.

Software developers remain focused on the individual components, and the operations engineer focuses on the system as a whole. They'll take a holistic approach to ensuring that the entire system is running reliably and functioning correctly. In turn, by spending less time on operations, developers spend more time on feature development.

The operations engineer also serves as a team utility player. While their primary role is ensuring the reliability of the team's services, they will be good enough to offload, augment, or fill in for other roles when needed.

There are tons of definitions and implementations out there for the word DevOps, but this new team formation is, to me, the greatest expression of that word. DevOps is the people, process, and tools for achieving outcomes and value.

We've long realized the value of collaboration and cross-functional teams in promoting success. To me, dissolving the operations team and adding its members to a cross-functional team aligned around a problem domain is the most efficient means of delivering value.

How much closer can you make collaboration than placing people on a singularly aligned team? Serverless will help us to fulfill what many of us have been trying to achieve for years.

Chapter 4:
Why Serverless?

"I DON'T WANT TO JUST OPERATE SYSTEMS ANYMORE."

So far, I've presented serverless only as a disruptive technology in the world of operations. It's going to change how operations functions and how we apply our existing skills. It's also going to require us to learn new skills.

In only that light, serverless seems like something to be afraid of. But it's not! It's something that should be embraced with optimism because of the changes it brings.

Let's discuss why so many are excited about the effect of serverless on operations. In this chapter, I'll talk about what drives many of us (including myself) in this field, where that drive has been lost, and why serverless brings it back.

Getting Into Ops

I got into operations for two main reasons: I enjoyed building things and I enjoyed solving problems. Unlike many people in the operations profession, I was not responsible in most of my jobs for "operating" software built by developers. Instead, I usually worked on infrastructure and service delivery teams.

For most of my career, particularly in the beginning, my two motivators were directly linked and operations was very enjoyable. There was a problem that bugged me and I built something to alleviate that problem. I was either solving a problem of mine or solving a problem that helped the teams I served.

Over time, though, I started getting bored. While the technology available to me has changed, the problems haven't. Whether it was delivering a virtual machine on VMware or EC2 instances in AWS the problem was still, "How do I deliver compute to a waiting engineer?"

Similarly, whether it's building an application packaging, deployment, and runtime platform or choosing to containerize applications with Docker, these two problems are largely the same: "How to I bundle an application and its dependencies, and deploy it to a standardized platform to run?"

I was tired of keeping up with changes to operating systems, too. The host operating system is largely a means to an end for me. I prefer to spend very little time logged into an application host. Most of what I need from a host — logs, metrics, etc. — should have been shipped to another system that I could then interact with. Changes to network device naming, increasing systemd complexity, or replacing a standard UNIX command with some new utility may benefit some people, but for me these things largely get in the way. They require engineering or effort on my part to keep up while providing little to no value.

What's even more frustrating to me, the problems operations engineers are asked to fix across most organizations are largely similar. This has led to organizations differentiating themselves based on their technical stack and technical solutions to attract talent. And that technical stack may not even be the right choice for their problems, current maturity, or scale. I see startups deploying Kubernetes to run only handfuls of containers, for example.

> **"We're building a service mesh on top of Kubernetes. That's a great reason to work here!"**

At this point in my career, the trendy option of building and operating a Kubernetes platform for container management is simply not appealing work anymore. It's just the third iteration of a problem I've solved more than once before. And the sheer number of operations jobs asking for the same problems to be solved means most organizations fail to stand out. (And other differentiators like culture are hard to evaluate until you already work there.)

How Does Serverless Make That Happen?

Offloading operational work to public cloud providers is highly appealing. It lets us shed undifferentiated work that we've been doing repeatedly across our careers and focus on serving people and providing value. While at first I was worried that serverless would eliminate all of my work as an operations person, I eventually realized that would not be the case.

Why? Because most of us work in organizations where there's more work to be done than there is capacity to perform the work. Serverless doesn't result in NoOps, where the need for operations goes away, but instead what we might call DifferentOps.

The effects of serverless on operations can mostly be characterized by

- **Greater emphasis higher up the the technical stack**
- **Time and effort freed up for tasks we couldn't get to before**
- **More time to solve business problems**

Let's briefly explain these and why it makes serverless exciting as an operation person.

Moving up the Tech Stack

I've managed Linux hosts for a long time and it's fairly routine, except for when network device naming changes or a standard decades-old command is replaced for some reason. Whether it's working directly on hosts, building machine images, configuration management, or figuring out how to deploy hosts, I've done it and I've done it for awhile, which now makes the work relatively routine and boring.

Take the infrastructure and the associated work away, and what do you have left?
You still have much of the same work, but now you're performing it against the application. If you're monitoring and troubleshooting hosts today, then tomorrow you're monitoring and troubleshooting applications. If you're tracking host vulnerabilities and patching software today, then tomorrow you're tracking and patching application and dependency vulnerabilities. And so on.

Many of us in operations are used to using tools like strace, sar, and vmstat to observe and debug what the host operating system is doing. This also means, as operations people,

we're going to have to dig into code. We'll have to understand debuggers, profilers, and tracing. Personally, I've always wanted to learn those skills. And tomorrow I may be on a team with an experienced application developer who can help me.

The work is new and the work is different, but the fundamentals are the same. And while much of the work may be boring and tedious to an experienced developer, it's work that I can tackle with the enthusiasm of a junior engineer excited to master new skills.

Assessing and Improving Software

We should acknowledge there's always more work to be done in our infrastructure. That means the free time we gain from not doing much of the operations and infrastructure work can be used to improve the software we're responsible for.

But many of us have a hard time picturing new work to do. That's because we have become stuck in a rut, doing mostly the same things and solving the same problems day after day and job after job. But serverless provides an opportunity to break free from that rut.

Much of the new work we can do after service delivery is to continually assess our applications for reliability and resilience to failure. This new work starts with planning game days in your organizations. These are fire drills to assess our preparedness and response to failures in the environment.

A more rigorous discipline call chaos engineering is also developing, where teams take a disciplined and scientific approach to testing for failures. With chaos engineering, you form a hypothesis of what and how systems fail, perform controlled experiments to test whether you were correct, and then from the data collected learn and apply your new knowledge to improving a system.

There's also a new push to start performing software and system tests on production instead of staging servers. The best tests of a production system are performed on that production system and not a staging environment that is mostly similar and under significantly less load. But to do that, you need to have your failure preparedness plans already in order and good knowledge of how your systems may fail.

I should point out that just going serverless isn't going to magically make you able to perform these practices. But you should have the time to work on the people and process in your organization so that you can adopt these practices successfully. Serverless here provides an opportunity to make our organizations perform at their best.

Solving Business Problems

I've spent a bit of time in startups, and it's dramatically altered my thinking about engineering and what's really important over time. Now, I'm more interested in solving business problems and the growth of the organizations I work for.

At my first startup I was introduced to product metrics and terms like adoption, retention, and churn. Development teams released products and features, and their work drove metrics, which ultimately drove revenue. That's what is so interesting to me about product pods and feature teams aligned around business problems. Adding a feature that delights users, fixing a bug that frustrates them, or any number of other product changes shipped had a measurable and noticeable effect on the organization.

But not a single customer really cared if we ran on-prem or in the cloud, ran in AWS or on OpenStack, or whether we were deploying Kubernetes. My work on an operations team never budged the metrics that were most important to the organization. Serverless,

and the potential repositioning of operation roles in an organization, provides an opportunity to have a direct impact on the organizations we work for by increased emphasis on solving business problems over technical problems.

Today, I'm very interested in the intersection of solving business problems and engineering through the use of product engineering concepts. You're starting to see product engineering concepts creep into operations already.

You may have heard the phrase "**product, not project**" more recently. This mentality involves solving problems, and engineering, iteratively. You start by identifying a problem and delivering small solutions quickly, then you measure the success of your solution to determine whether to continue your effort or take a new approach. This problem-solving and engineering is much more difficult than the two hardest problems in engineering, cache invalidation, naming things, and off-by-one errors.

Let's also talk about judging our success. Projects are judged by benchmarks like schedule, budget, and technical correctness. An organization largely judges this on their own. A project is a success or failure because the organization declares it a success or failure.

Products are judged very differently. They're judged on the previously mentioned metrics like adoption, retention, and churn. To be successful, a product has to solve a problem for which there is a demand and in a way that will make the user happy. Success is judged by the fickle whims of external arbiters. This makes attaining product success far harder than almost any technical problem out there. This is a whole new level of hard problems. And these are the challenges that I want to face.

You don't have to work in a startup or product company to adopt this product-not-project mentality. Even as an internal engineering or service team you can still work this way.

After you deliver a service do you follow up with users? Even if they haven't complained? Do you check that people are using your service? Do you check that they're satisfied? Do you check that their problem has been solved or improved? There's so much more we can do to ensure we're solving real business problems and having an impact on our organizations.

Dropping infrastructure operations work by going serverless and joining a product pod lets me focus on solving problems that are relatively unique to my organization and have a direct contribution to the growth and success of my organization. And these types of problems provide a level of difficulty and challenge that is far harder than anything I've experienced in my career.

I Don't Want to Operate Systems

I've reached a point where I need to say this.

"I don't want to JUST operate systems anymore."

I'll just come right out and say this. I didn't get into this profession to operate systems. The work of operating systems is obviously important, and still is with serverless, but I want to reduce my time spent on that work. I want to do other things that contribute to what I really enjoyed about operations. When I express this sentiment out loud I find people coming forward to express similar feelings. They're less interested in operating systems and more interested in providing value and serving people.

Chapter 5:

The Need for Ops

"OPERATIONS NEVER GOES AWAY . . ."

Now let's start to talk about what operations people bring to a team. These skills are going to provide greater depth to a product pod's overall depth of skills. But, there's also a skill, coding, that we're going to need if we haven't yet developed it. Then from there, let's talk about some of the areas of operations for serverless applications and how by being serverless the work evolves.

Essential Ops Skill Set

In order to be effective in their new role, what skills will be required of the operations engineer? There's a variety of skills that operations people have acquired over time that will serve the team well. But there's also skills we'll need that many of us are weak on.

Let's break this down into two sets:

1. **Skills currently possessed by the typical operations engineer today.**
2. **Skills many operations people will need to level up or possibly begin acquiring.**

The Skills Operations Engineers Bring

Let's start by running through the skills operations brings with their expertise. Historically these have been the primary skill set of operations engineers.

Platform/Tooling Understanding

Every engineer develops expertise in the platforms and tooling they use regularly. Frontend engineers develop expertise in JavaScript and its frameworks (or they invent their own). Backend engineers develop expertise in building APIs and communicating between distributed backend systems.

As operations engineers, we develop expertise in platforms and tooling. In an environment running on AWS, we develop expertise in automation tools, such as CloudFormation and Terraform. These are tools with a steep initial learning curve.

We also have expertise in aspects of AWS; for example understanding the idiosyncrasies of different services. Lambda, for instance, has a phenomenon called "cold starts" and its CPU scaling is tied to function memory allocation.

Systems Engineering

The systems engineering skills are present when designing, building, operating, scaling, and debugging systems. As operations engineers, we're required to develop an understanding of the overall system we're responsible for and, in turn, spot weaknesses and areas for improvement, and understand how changes may impact a system's overall reliability and performance.

People Skills

Finally, there are people skills. While computers are fairly deterministic in their behavior, people are not. That makes people hard and we need to stop calling the ability to work with them a "soft skill."

Much of the work of operations engineers is figuring out the actual problem a person is trying to solve based on the requests people make to them. A good operations teams is a good service team. They understand the needs of those they serve. People skills and the ability to get to the root of a user's problem will be complementary to the product manager's role on the team.

The Skills We Need

Here are the skills we'll need to level up on or acquire wholesale. Warning: because of the haphazard evolution of the operations role in different directions because of different definitions of DevOps, there is an uneven distribution of these skills.

Coding

There's no avoiding the need for operations engineers to learn to code. And they're going to need to learn the chosen language(s) of their team. Whether it's Python (yea!), Go (looks interesting), JavaScript (well, okay), Java (ummm...), or Erlang (what, really???), the operations person will need proficiency.

Keep in mind the ops person serves as a utility player on the team. They're not there to be just another developer. Being proficient in a language means to being able to read it, code light features, fix trivial bugs, and do some degree of code review. If you're quizzing operations job candidates, or even existing operations employees, on b-tree or asking them about fizz-buzz solutions, you're probably doing it wrong. If you're asking them to code a basic function or read a code block, explain it, and show potential system failure points, however, then congratulations! You're doing it right.

For those who are code-phobic, don't fret. You won't have to snake your way through the large code base of a monolith or microservice. By isolating work into **nanoservices**, the code should be easier to comprehend.

Areas of Responsibility

Once an operations engineer joins a product pod, what do they do day in and day out? What will their job description be? What tasks will they assume, create, and prioritize? We need to be able to articulate a set of responsibilities and explain the role. If you can't explain your job, then someone else will do it for you. And that could mean your job is explained away with "NoOps" as its replacement. Let's lay out some areas of responsibility operations people should be involved in when it comes to supporting and managing serverless systems.

Architecture Review

Serverless systems are complex systems and in order to remain manageable they need to be well architected. The ability to quickly deliver a solution also means the ability to quickly deliver a solution that looks like it's composed of duct tape, gum, and popsicle sticks. Or, it looks like something out of a Rube Goldberg machine.

Just take a simple task like passing an event between the components of a serverless system. Most people will do what they know, and and stick to what they've previously done and are comfortable with. But even such a simple task can be solved in a variety of ways that may not occur to an engineer. When do you use SQS over SNS; or even both in conjunction? Maybe a Lambda function fanout? Perhaps event data should be written to persistent storage first, such as S3 or DynamoDB, and use events from those actions used to trigger the next action?

> **"Generally we should use SQS or Kinesis instead of SNS when writing to DynamoDB in this sort of application. They allow us to control the rate of writes. With SNS we're more susceptible to failures due to DynamoDB autoscaling latency."**

What's the best choice? The problem you're trying to solve, requirements, constraints on you will dictate the best choice. (As an aside, I'm very opinionated about Lambda functions fanouts and will ask why you feel the need to handle function retries yourself. There are valid uses cases but IMHO, few.) Serverless systems introduce significant amounts of complexity and that can't be denied. Couple that with the multitude of ways that even a simple task can be solved and you have a recipe for unmanageable systems.

The operations person should be able to explain what architecture patterns are most appropriate to solve the given problem. Start by ruling out patterns and then explaining the remaining options and their pros and cons. And, explain your reasoning to other engineers so they can make the best choice on their own the next time.

Reliability

Metrics, monitoring, observability, and logging; as operations engineers we'll continue to own primary responsibility for these. What will change is the issues we'll be looking for in these areas. Host CPU and memory utilization won't be your concerns. (That is beyond the cost associated with higher memory invocations and the impact it has on your bill.) Now you'll be looking for function execution time length and out of memory exceptions.

It will be the responsibility of the operations person for selecting and implementing the proper tooling for solving these issues when they surface. If a function has started to throw out of memory exceptions, the ops person should be responsible for owning the resolution of the issue. The solution may be as simple as increasing function memory or working with a developer to refactor the problematic function.

There's also a phenomena known as "cold starts" on Lambda that occur when a function's underlying container (there aren't just servers but also containers in serverless) is first deployed behind the scenes. You'll need to observe their impact on your system as well as determine whether changes need to be made to alleviate them.

"While this service sees a certain percentage of cold starts, the data is not required and consumed

by the user in real time. Cold starts have no actual user impact. Our engineering time is better spent worrying about the smaller amount of cold starts in this other system that is user interactive."

Your operational expertise and knowledge of the system as a whole will help inform you of whether these need to be addressed since cold starts aren't necessarily problematic for all systems.

This are just some of the things that can go wrong with a serverless system. This isn't by any means an exhaustive list of what can go wrong of course. And often, something we as ops people are tasked with responding to is something we didn't even initially think of. So instead of continuing to enumerate all the possible ways that systems can fail, let's spend more time on how to catch and respond to failure.

Keep in mind, even the simplest serverless application is a distributed system with multiple services that are required to work with one another. Your systems engineering knowledge is going to come in very handy.

Performance and Maintaining SLOs

As your application incurs greater usage, can you still maintain your established **service level objectives (SLOs)**? What about when you're tasked with performance tuning a running system to meet a new SLO target, what do you do?

There will be work at rearchitecting systems by replacing individual services with different ones that better match new needs. Or maybe the system as a whole and its event pathways

need rethinking? Different AWS services have different behaviors and characteristics and the right choice initially may no longer be the best choice.

> **"We're going to need ordered messages and guaranteed processing here to accommodate this requirement. That means we'll need to swap SNS for Kinesis here."**

Evaluating code changes will also be a part of achieving new scale and performance goals. You'll start by using the growing performance and reliability tools in the space to find and refactor inefficient functions.

You may even find yourself looking at rewriting a single individual function in a different language. Why? Because at a certain point in increasing Lambda memory, an additional CPU core becomes available and your current chosen language may not be able to utilize that core. The operations person will be responsible for knowing facts like this. (The ability to rewrite a single nanoservice function compared to rewriting an entire microservice is pretty cool!!?!?)

Scaling

We're used to scaling services by adding more capacity in some way. Either we scale a service vertically (giving it more CPU, RAM, etc.) or we scale a service horizontally (adding more instances of the service.) When it comes to scaling work we're used to either migrating a service to a new larger host or adding additional hosts.

But serverless has scaling, typically horizontal scaling, built in though. That's one of it's key characteristics that makes something serverless! So what's left to scale.

The ability to rapidly scale creates its own class of issues, such as thundering herd problems. How will a change in scale or performance affect downstream services? Will they be able to cope? A lot of what's been discussed in the previous sections with regard to performance and reliability are actually by products of serverless systems' ability to scale so easily.

You're not responsible for scaling individual services, you're responsible fo scale entire systems now. Someone will need holistic view and responsibility for the system as a whole so they can understand how upstream changes will affect downstream.

Security

Finally let's touch on security. There's a regular debate among engineers about whether giving up control over security responsibilities makes you more or less secure. I take the argument that offloading those responsibilities to a reputable cloud provider such as AWS makes you more secure. Why? It's because they can focus on issues lower level in the application stack which frees you to focus your attention elsewhere? What does that mean in practice?

To begin, you're no longer responsible for host level security. In fact let's take that statement a step further. You're not responsible for virtualization hypervisor security, host level security, or container runtime security. That is the responsibility of AWS and AWS has proven to be effective at that. So now you're not responsible for this, what are your responsibilities?

To start, there's the cloud infrastructure controls. This mostly boils down to access controls. Does everything that should have access to an AWS resource have access to it, and access

is limited to only those with that need? That starts with ensuring S3 buckets have proper access controls and you're not publicly leaking sensitive data. But that also means ensuring a Lambda function that writes to a DynamoDB table only has the permission to write to that DynamoDB table; not read from it for write to other tables in your environment.

If AWS is handling the bottom of your technical stack where should you be focusing? Up towards the top of course! That means greater emphasis on application security. Don't have to monitor and patch host vulnerabilities anymore? Good, go patch application vulnerabilities such as those in your application's dependencies.

Finally, You've heard about SQL injection before where SQL queries are inserted in requests to web applications, resulting in the query's execution. Well now think about event injection, where untrusted data is passed in Lambda trigger events. How does your application behave when it receives this data? There's a wealth of attack techniques to be rethought and see how they apply to serverless applications.

Operations Doesn't End Here

This isn't the entirety of all the operations work that will be necessary to build and run serverless systems but it's a start. In coming chapters we'll expand on some of the skills mentioned as well as some of the areas of responsibility.

Chapter 6:

The Need to Code

"I TAKE JSON FROM ONE API AND I SEND IT TO ANOTHER API."

Let's talk about diving into, and even learning how to, code. This is not an area we've traditionally been responsible for, but the overwhelming trend in operations the past several years is the expectation to deal with code to a greater degree. With serverless, all that's left is code. Skills in and around code become a minimum requirement. We're going to talk about some basic expectations and responsibilities for operations people around code and why we shouldn't be particularly afraid.

The Need to Code

Operations will have to take a greater role in application code, and expect to be required to contribute throughout the entire code lifecycle. That includes writing, review (both before and after writing), testing, and the build and deploy process. That range might seem intimidating, but it's actually helpful. For those of us who are less advanced in coding, it gives us a variety of ways to contribute as we work to strengthen our skills.

The exact level and type of participation by the operations person will differ depending on the team and organization, of course. Different organizations and teams will have different needs. If your organization doesn't expect or highly value coding among operations engineers, there will inevitably be a more limited coding scope than in an organization that does expect and value the skill. However, that will only last so long as operations engineers will (and should) be expected to level up to increase their effectiveness in this area.

You might be tempted to say, "I've gotten along this far without coding, I'm sure I'll be just fine when we switch to serverless." You would be wrong.

I've tried to imagine a way in which we can remain a productive pod team member without coding skills, and I just can't. When there's an issue today, you as an operations person can start by investigating the host or other infrastructure issues, and then hand application code investigation to a developer.

But there is no host for you to investigate with serverless. Your cloud infrastructure is less complex. The majority of what's left is code, and most issues will stem from that code in some way. You're not going to have enough work to do, and if you don't have enough work to do your organization is going to question why you're there.

I don't like being negative and I'm sure what I've just said, particularly concerning job security, may bother some people. My goal isn't to offend, but to motivate. I say all this as someone who a few years ago found their operations job in a precarious position because of their limited coding skills. I've gone through this once before and I've worked to ensure I don't again. I'd like to keep other people from that same experience.

Serverless Makes Code Accessible

I don't want to leave people with a feeling of dread about their careers because they now need to learn how to code. You may be scared of coding based on past experiences. That's totally understandable because it's hard.

But what's great about serverless is that it makes coding more approachable. Many of us have stared at a large code base, gotten lost, and given up. I have been there myself and admit coding has been the skill area hardest for me to advance in.

One of the experiences that attracted me to serverless was the simplicity and approachability of the code involved. Take a simple non-serverless HTTP microservice today, written, say, in Python and using the Flask framework. There's quite a bit of code involved before you ever get to the business logic. You have an application entry point that potentially reads from a configuration file. You need to configure where logs are directed to. Then, you need to establish application routes or endpoints. And finally, you're adding business logic.

For an experienced coder, that may seem trivial. But for the less experienced, that overhead leads to intimidation and anxiety. There's a lot going on around the particular code you're interested in.

Compare that with a serverless function on AWS Lambda, where you define an entry point to your business logic. Your HTTP routes or endpoints are defined on API Gateway with which this function is associated. Your logging is fairly simple because it's often just to standard output so it's picked up by CloudWatch.

When you're looking at a serverless function, you're mostly looking at only the essential code and not extensive amounts of setup code. I've found this focus makes serverless functions easier to debug and understand. I don't need to snake through an extensive codebase. When I'm investigating an issue, much of the code I need to examine is right there in front of me.

If you've had bad experiences with code before, give serverless a chance. You may find your experience different this time around. You may even find the confidence to start building this lagging skill of yours. It happened to me.

Coding Required of Ops

To start, the operations engineer should be proficient in the languages in use by the team. You should be capable of fixing light bugs. While carrying out your reliability responsibilities and investigating errors, you shouldn't stop at determining probable cause and filing a ticket. Always go further into the code involved. That means isolating potentially problematic code, at least to the function instance, and fixing minor issues.

"This code expects an int, but the data actually contains a string representation of an int. Let me handle this."

If you can't fix the bug, you should write a thorough bug report. Guide the developer who will fix the issue as close to the problem source that you found in your investigation. Here, communication skills will be of the highest importance. So let's dive deeper into what this work looks like.

Code Review

To start, there's code review. We should be bringing our knowledge and perspective of how the platforms we use work and think about the code as a part of the greater system. I was fortunate enough a few years ago to work with a very good operations-minded developer. They taught me how to write more reliable code by teaching me during review the ways in which cloud systems fail.

It's not that I didn't already know those things, though. I just made assumptions that operations would succeed and often didn't account for failure. I was guilty of putting aside my operations and distributed systems fallacy knowledge as I coded. Assuming the network was reliable was my most common mistake. Review lead to the team building more reliable services.

> **"This function should have retries because we may fail at the end here. However, you can't safely retry the function due to this earlier spot. We'll end up with duplicate DynamoDB records."**

I think there's also a good, practical reason for operations engineers to be involved. We've mentioned the inexperience of many operations people with code. This gives them a chance to pair with a developer and learn.

Testing

Software testing is a step beyond code review. You're not just evaluating how software works but whether it works, too. Best of all, to test whether code works you have to write more code. For someone inexperienced writing code, these coding tasks should be approachable, and the work provides practical experience.

But shouldn't a developer be writing their own test? Of course. But how many code bases do you know with 100 percent test coverage? The reality that is software development leaves a gap for us operations people to fill.

Bug Fixes and Basic Features

Ideally, we should eventually level up to be able to code bug fixes and basic features, as well. The only way to stay sharp in a skill, and for operations engineers to level up in this area, is to do some of the work. Work that could be given to a more junior engineer could instead be assigned to the operations engineer.

> **"I can create a REST API with an endpoint that ingests the data from this service's webhook, enriches the data, and passes it onto this other system."** ***— Inspiration From Alice Goldfuss.***

In this capacity, the operations engineer provides extra development capacity. As your coding ability improves, you can be called on to augment the team's output when deadlines are tight or workload has become too big.

Operations First

One problem I see, which I expect will generate much friction, is developers or organizations incorrectly evaluating the coding skills of an operations engineer. Rather than judging operations engineers on writing the fastest, most efficient, or pedantically and subjective "best" code, they are there to help the team produce the most reliable code.

As only a part-time software developer, an operations engineer should be treated like a junior developer. You should be writing, clear, manageable, and reliable code that solves a defined problem. Beyond that, however, the skills required are that of a more senior software developer.

Work requiring a senior developer should go to a senior developer. Otherwise, the operations engineer will be set up to fail, and the team as a whole will fail to achieve what other more cohesive teams can.

Chapter 7:

The Work Operating Serverless Systems

". . . OPERATIONS JUST CHANGES."

We've covered some of the areas where operations knowledge is needed with serverless systems, including reliability, performance and architecture review. But what is the actual operations work involved? Let's walk through some practical areas where an operations person should be involved.

What Will Ops Do?

The work described in this section is well suited for an operations engineer. It isn't everything, of course, but it should give you an idea of an average work day. Use this as a base, and then expand the scope of responsibilities as you identify other work to be done. A lot of what you read will resemble work you're already doing. This is why we sometimes call serverless operations "DifferentOps".

Systems Review

Let's start with system reviews. Throughout the service delivery process, the operations engineer should be involved to ensure the most appropriate infrastructure decisions are made. If you've started your DevOps transformation, then you should be doing that already. This is what I mean by tearing down the silos and refraining from throwing engineering projects over the wall.

Take time to review your team's systems throughout all states of the delivery lifecycle and look for potential issues. Use your knowledge of operating systems and experience in seeing things go wrong in order to find potential problems and suggest solutions. Educate the rest of your team so they can make better decisions in the future, or at least know the questions to ask.

System Monitoring

Let's start with the basics of reliability: monitoring. As you look at the architecture of a serverless system, you should be looking for how it might fail. Or, if this is after

an incident, you should be looking for the reasons behind a service failure. If there are states the system should not be in, then you want to know about them.

You should be doing your best to assure that when a system reaches an undesired state, an alarm like that from **AWS CloudWatch** is triggered. You should be doing that already, but what's unique to serverless?

AWS CloudWatch alarms are just AWS resources, and that means you can manage them with CloudFormation. Don't divorce your serverless code and system configuration from your monitoring alerts. Don't have one in source control and the other manually configured in the AWS console. If you're using CloudFormation, AWS SAM, or Serverless Framework, then add **CloudFormation resources for CloudWatch Alarms**.

You can't be expected to review every single new service and put your stamp on it to ensure proper monitoring. In fact, that's probably not going to fly. Making yourself a gatekeeper to production removes one of the key values of serverless, which is speed of service delivery. Instead, enable the rest of your team to do that work on their own. As mentioned later in **Build, Testing, Deploy, and Management Tooling**, the rest of your team may not be as fully versed in your tooling, particularly CloudFormation syntax.

One reason I like Serverless Framework is the ability to extend it through plugins. Take a look at this plugin, serverless-sqs-alarms-plugin. It simplifies the configuration for adding a multiple SQS queue message size alarm. If you want to enable your teammates but they struggle with CloudFormation, write more plugins to abstract away the complicated parts and make configuration simpler for them.

Function Instrumentation and Observability

A close cousin of monitoring is observability. **Observability is the ability to ask questions and get answers about your system and its state**. While monitoring is raising the alarm on known bad states, observability is providing enough detail and context about the state of the system to answer the unknown.

A key facet of observability is instrumentation. Instrumentation is the collection of data about your system that lets you observe it and ask questions. Ideally, the function's author should be instrumenting a function. They're in the best position to do this work because they should have the best understanding of what the function does.

But observability is all about dealing with unknowns, and that means the proper instrumentation to answer your question may not be in place. If that's the case, you as the operations person should be ready and able to instrument a function with additional code. If you need to add additional measurements, trace points, labels, etc., to solve a problem, then you should be in a position to work with your observability platform's SDK to do so.

I bring this up because you should be expected to do this work as a part of your operations reliability responsibilities, but also because for those coming from a background with less code experience, this is a good start for getting familiar at working within a code base. It's going to force you to read code to find the relevant parts you need to observe, add additional code, and go through the proper process for deploying your change. It might seem small to those in operations who are used to working with code, but it will probably be a leap for many ops people.

Service Templates

A complaint I've heard regularly about serverless is the amount of configuration overhead incurred. Each new project requires AWS CloudFormation or Serverless Framework configuration.

I have a standardized project layout I use to make finding what I'm looking for quicker and easier. Services require test coverage and a CI/CD pipeline. As I established new best practices and standards I wanted to adhere to, the more work it became to create a new service. Soon, the amount of overhead in creating a new project became frustrating.

My solution was to investigate **Serverless Framework templates** used during project creation time. I realized I could create my own templates to incorporate the standards I expected my services to meet. I now have a template for **AWS Lambda with the Python 3.6 runtime**.

This template saves me an incredible amount of time. New services are initialized with a standard project skeleton. A basic Serverless Framework configuration is created. A project testing layout is created and configuration is created. A TravisCI configuration is created. This might seem simple, but it's a way for you to ensure that people are creating new services in a standardized way.

Eventually, I want to take this a step further and create a library of referenceable architecture patterns. That library would demonstrate different infrastructure patterns, explain their pros and cons, and, finally, explain when each pattern should be employed. By creating this library I'm empowering the rest of the team to make better decisions on their own. The better we can empower developers to make good decisions up front, the less time we spend in meetings correcting errors.

It would also show the configuration necessary to deploy the given infrastructure pattern. Much of my time is spent in AWS CloudFormation docs trying to remember how to do something I've already done before. Imagine the amount of time I spend in them figuring out how to deploy infrastructure I haven't deployed before. Then image how frustrating it must be for a developer who doesn't have the same level of experience.

Alleviating this frustration and helping developers deliver more easily is a part of our job and this is an area we can greatly help a team with.

Managing SLOs

The operations person will be responsible for ensuring that service-level objectives (SLOs) are defined and that they're being met. What metrics does a service need to meet in order to be considered functioning normally? When they're out of bounds, what does the team do?

The first step to managing SLOs is creating them. Your SLOs aren't arbitrary values. They represent the expectations and needs of your organizations and its users. To establish these means understanding those needs and translating them into engineering. You should be working with the team's product manager to understand what the business is trying to achieve. You should also be working with a UX engineer, if you have one, to understand the needs of your user. Remember these important words.

> **"9s don't matter if users are unhappy."**

Keep in mind that SLO targets may change over a system's lifetime. For example, your organization may determine a new SLO target for a service. Why? Maybe your organization has determined that decreasing page load time increases user retention. To achieve that,

the variety of services involved in rendering that page, including services owned by your team, need to meet a new target. People are going to have to navigate the affected systems and look for areas of optimization to meet the new targets.

Also keep in mind that as usage scales, you may begin to expose limitations in your systems. A system that serves 10,000 users may not be able to serve 100,000 users and still meet its targets. We'll come back to this, though.

When you have SLO targets defined, then you must actively follow them. Don't wait for services to cross your thresholds in production before you react. Be proactive. In production, keep tabs on system performance metrics compared to target metrics. If a system is getting close to its threshold, then budget some time to have a look at areas of improvement. In your CI pipeline before you even reach production, look at writing and include performance tests, as well.

Usage Scale Point Management

I mentioned earlier that a system that serves 10,000 users may not be able to serve 100,000 users and still meet its SLO targets. You might be tempted to move the goalposts for your system so it can support 100,000 users right from the start. Premature optimization, however, isn't the answer. Know both your expected usage scale targets as well as the scale points at which your system can no longer meet your SLOs.

This starts in the planning phase. The team's PM should have realistic target usage metrics. It's tempting to believe your new application or feature will be a runaway success that everybody will want to use, but that's rarely the case.

Target usage projections aren't exactly a science, but you have to start somewhere. If you only expect to have 2,000 users by the end of the year don't architect a system to support 100,000 users from day one. By the end of the year you might find that you failed to achieve the expected growth and what you've built is going to be scrapped. Any extra engineering to support 100,000 users would not have been worth it.

With target usage established, stress your system as it's being developed and after it's in production. Ensure you've left runway for growth, too. As you're closing in on your break points budget time to refactor those points.

How do you find those points? Try a tool such as **Artillery**. Run controlled tests to find at what usage scale your applications breaks and how it breaks.

The New Work of Ops

The work described in this section isn't exactly new. We've read blog posts and seen conference talks on this work. We may even know people who do this regularly. But the sort of organizations doing this work regularly are typically highly mature and advanced. These are things we should be doing but many of us are not yet.

This work is not serverless specific or a result of serverless technology itself. I mention these ideas because of the organizational change effects that serverless creates and the result of increased time availability. Serverless isn't merely a technical change but it has ripple effects on your organization. Use your new time wisely. When in doubt about what to do, look for areas of work you're not doing currently but feel you should.

Incident Commander Management

Let's start by stating what this is and what this isn't. As the operations person, you're not the first responder to every alarm. That's just a return to the "shipped to prod, now ops problem" mentality. Second, you're not the incident commander (the person directing the response) during every incident. That's going to breed a culture where people learn only enough about the systems they support until you take over.

Ideally, the operations person should become the individual who oversees the entire incident command process. Your role as the incident commander manager isn't serverless-specific, but a function of adopting the product pod model. As the operations person, you're probably the most experienced and qualified in this area.

Incident command doesn't start at the beginning of an incident and end at resolution or, better, a post-incident (post-mortem) review. New people need to be leveled up on how to be a proper first responder and incident commander. They should be trained on what to do before they wake up at 3 a.m. As the incident commander manager, be the person in charge of ensuring the rest of the team can find what they need to solve an issue. Additionally, the process should always be going through refinement.

Chaos Engineering

The new discipline of **chaos engineering** deserves a special mention. The cloud can be chaotic, and with serverless your placing more of your infrastructure into the hands of your cloud provider. Being prepared for failure, particularly failure that you can't control, will be increasingly important.

To start, what is chaos engineering? Chaos engineering is the discipline of experimenting on systems to understand weaknesses, and build confidence that a system can withstand the variety of conditions it will experience. It's an empirical approach to understanding our systems and their behavior, and improving them.

If you're no longer responsible for operating the message queue service in your environment, for example, then you have more time to understand what happens when it fails and improve your system's ability to handle it. Use your newfound time to plan events such as game days, where you test hypothesis and observe not just individual reaction, but team reaction, as well.

Chaos engineering takes times. It takes time to formulate hypotheses. It takes time to compose and run experiments. It takes time to understand the results. And it takes time to implement fixes.

But you should have that time now. Let concepts like chaos engineering and game days go from something you only hear others to things you can actually implement in your own environment.

There's Still More

This, of course, isn't the entirety of the work required to operate serverless systems successfully, but it is a start. In fact, cover security as well as the build, test, and deploy lifecycle along with financial management separately.

Chapter 8:

Build, Testing, Deploy, & Management Tooling

"SHIP IT, SHIP IT REAL GOOD."

What is one of the major obstacles we all face as engineers? Understanding our tools. The tools of our trade come with significant complexity. But as we grow to master them, these tools become significantly more powerful.

Your team's service delivery pipeline is an excellent place to take ownership and apply your skills. Not only is the work important, it's also an excellent way to automate manual tasks suggested elsewhere in this ebook. If you can catch and prevent issues during the service delivery process, you'll spend less time fixing them in production.

What Does Ops Bring?

Throughout my career, I've often found myself being the go-to person for explaining the functionality of tools. I'm not sure why, but I think it has a bit to do with my fascination with how the things around me work. I'm also perfectly comfortable with declarative DSLs, YAML, and JSON for automation and infrastructure tasks.

As a general rule, operations people have to learn how the systems they support work, even if they didn't build them. In fact, if your culture is still heavily siloed between development and operations, you're probably accustomed to learning how something you didn't have a hand in designing or building works.

For example, have a look at the service delivery pipeline in your organization and the variety of tools involved in getting a service into production. There's probably some sort of infrastructure automation and orchestration tooling. When someone needs to deploy a new service, how do you create new infrastructure for it? Are you currently a part of maintaining the Puppet/Chef/Ansible infrastructure? What about Terraform and CloudFormation?

These tools are usually a significant part of the service delivery pipeline, often playing a role in the entirety of build, deploy, testing, and management. Their importance to every phase of service delivery can't be ignored. And if people think they're going to go serverless without these automation tools, they're quite simply going to fail.

Next, look at your continuous integration (CI) platform. Considering just about every CI tool was written out of hatred and has lead to another tool people hate, I prefer to focus on the positive. I like CI platforms, even if they are imperfect and occasionally troublesome. Supporting the CI platform in my organization was an opportunity to scale myself through

automation. There's no way I was going to look at every change and put it through a checklist before it was deployed. That gatekeeper role just doesn't exist in an increasing number of organizations. But I could write checks to ensure people followed expected practices.

The need for this knowledge of tooling in the software delivery pipeline doesn't go away. Someone will have to take ownership of it and ensure that developers can deliver with minimal friction. That should continue to be operations people.

What Will Ops Be Doing?

Now that we've walked through the service delivery pipeline, what should you be doing in that process? How will you carry out your responsibilities? Let's discuss several areas.

Deploy and Management Tooling

Tooling like **CloudFormation**, as well as **AWS SAM** and **Serverless Framework** (both built on top of CloudFormation), can be complex. I find many developers are not fans of configuration and domain-specific languages. Your dev team, if left to their own devices, will hardcode assumptions and thwart a lot of the problem-solving CFN has built in through sheer frustration with the management tools. CloudFormation is actually quite flexible if you're familiar and understand how to use it.

If you're already doing extensive infrastructure as code in your environment, you're probably the established authority on tools like Puppet, Chef, Terraform, etc. In that position you guide people to use best practices with those tools to accomplish their objective. The same will be true with serverless tooling.

How to do everything in CloudFormation is not always immediately obvious. I've struggled several times when trying to build new services. For example, to build my first CloudFront distributed site — because I found I needed that to do HTTPS with a custom DNS name — took me several tries. Ultimately, I did it by hand and worked backward through the screens in the AWS console and the CloudFormation documentation side by side. Then there was the time I wanted to automate AWS Athena and found out I had to look at the AWS Glue documentation.

How do you assist your team with your chosen tools? First start by being available. Be responsive to requests for help in these areas. Take time to review your team's work and look for areas of improvement, as well.

> **"You don't need to define the S3 Bucket name in this template. CloudFormation will generate one on its own so we can use CloudFormation's Ref function to pass it to the Lambda function through an environment variable and then here in your code get the name of the bucket from that variable."**

Make sure you can provide people with example patterns they can look at. Serverless Framework and **AWS SAM CLI** can create new projects based on templates. Provide a template for what a RESTful web API should look like, what a single-page application hosted in S3 using CloudFront should look like, and so on. Having these patterns documented and findable will save your team time and make you more productive.

Next, improve the tools on the team. One of Serverless Framework's improvements over CloudFormation is its plugin capabilities. If the tool doesn't do what you want it to do, make it. Can you make your tools do some of your work, such as ensuring adherence to defined engineering standards, automatically? Just like writing Puppet facts and Chef knife plugins, developing Serverless Framework plugins is a handy skill.

I point to **serverless-sqs-alarms-plugin** routinely as a good example of the sort of tooling development operations engineers should expect to do. You establish an engineering standard that SQS queues should have an alarm to indicate when queue processing is not keeping up. A single CloudWatch alarm can be quite verbose. Serverless-sqs-alarms-plugin, however, allows someone to add multiple alarms with minimal configuration quickly. You've made it easier for your team to add alarms and, in turn, it's more likely they'll do the work.

Another area to look at is developer workflow. Remember, operations isn't just about operating systems. It's about enabling others to get work done. Look for common frustrations the team experiences and see if you can automate that work. Hands down, my favorite Serverless Framework plugin is **serverless-python-requirements**. Why? Because it automates away the management of bundling my Python dependencies. It saves me so much effort.

Testing

Now let's talk about testing for serverless. There's a lot of focus on local testing of serverless applications and functions. For instance, there's **LocalStack**, which will let you run mocked versions of AWS services locally. But I think the desire to run services locally is a holdover from the pre-serverless development days when ops ran local VMs or docker containers to test against.

Why do people insist on full-featured local development environments? One reason is feedback loop speed. People want to find bugs quickly, which is completely understandable. A CloudFormation deployment can take time, which breaks concentration.

A second reason people want a local development environment is because they haven't been allowed to have their own in the cloud. Imagine the cloud provider bill if every developer was allowed to run a variety of VMs in your development environment. Virtual machines and containers made it possible for people to run local instances of the services they needed to work with.

One of the key characteristics of serverless architecture is not paying for idle time. That means the cost reasoning for why people need local instances should no longer apply. My own personal workflow involves writing unit tests with the **Python module Moto**, which mocks AWS services. I do that to catch the most basic errors before deployment. After unit tests pass, I deploy to my personal environment in AWS immediately to run integration tests. It has saved me a lot of time and hassle by eliminating the need to test everything locally.

Since you're no longer paying for idle capacity, exploit this new characteristic of your systems. Your developers should have their own cloud deployment of each service they need when they need it. As an operations person, don't spend your time trying to figure out how everyone can fully test their serverless applications locally. Do spend your time figuring out how to enable everyone to make as much use of your cloud provider and infrastructure as possible. If you'd like to see exactly how I approach testing and debugging, then read this: **AWS Lambda & Serverless Development - Part 2: Testing & Debugging**

Continuous Integration

There will be testing and deployment work, just as there is today. Your continuous integration tool is your friend. It is an automated and scalable version of you. In addition, testing, particularly integration testing, is going to become of increased importance with serverless. As you break down services into smaller, independent pieces, you're going to increase the chance of breakage.

Develop your checklist of things to look for in a project, then automate that checklist. If you've established that all SQS queues need CloudWatch alarms, then write a check. If an SQS queue is found in a project's configuration, the check will look for corresponding CloudWatch alarms and fail if they do not exist. You can find and prevent so many issues just by automating yourself during this stage.

Don't look at your CI platform as just a place to run a service's unit and integration tests, use it as a place to run your own checks, too.

Deployment

Once tests have passed, it's time to deploy new code. This is one of the less developed parts of the serverless software delivery process. If you're on AWS and using CloudFormation, or tools built on top of it, then your deployment tool is CloudFormation. Just keep in mind that CloudFormation doesn't have built-in capabilities for things like canary releases, blue/green deployments, or rolling releases.

I'm going to start by saying this: Blue/green deployments, canary releases, A/B testing, rolling releases, etc. are all good practices. But I've seen more than a few organizations

that either lack those capabilities or they exist in a very rudimentary state, and those organizations manage to function quite well. By delivering small changes, testing those changes extensively, and rolling forward quickly when issues arise, organizations manage to be successful without those capabilities.

I would not let the immaturity of those patterns with serverless hold you back from starting with serverless. I give this advice not out of recklessness or carelessness but out of acknowledgement that there's often a wide gap between best practices and what organizations actually do.

The good news for those of you who are interested in the topics around more robust deployments and reducing issues and errors on deployments, there's a lot of work to be done in this area that can keep you busy.

Closing

Delivery of serverless systems is going to keep you as an operations engineer very busy, particularly in the early days of adopting serverless. And for a variety of reasons, this is a great area for us to apply our skills. First, moving from Puppet and Chef to AWS CloudFormation, AWS SAM, or Serverless Framework is a logical progression in tooling. Additionally, managing the delivery pipeline allows us scale ourselves through automation. It gives us the ability to insert automated checks that ensure a certain level of quality that meets our standards leaves for production.

Chapter 9: Security

"GLAD I CAUGHT THAT OPEN S3 BUCKET OR ELSE WE'D BE THIS WEEK'S WINNER OF THE LAST WEEK IN AWS S3 BUCKET NEGLIGENCE AWARD."

Serverless, when using a public cloud provider, means offloading certain security tasks lower in the application stack to the provider, which in turn frees you up for tasks higher in the stack. When dealing with security, we usually try to work from the bottom up when securing an application stack, and rightly so. You can't secure an application properly without its underlying layers being secured properly. There's often so much work at the lower layers, however, that we don't have the time to secure higher layers adequately.

What serverless provides us with is an opportunity to offload much of the work at the lower end of the stack and, in turn, frees up time for us to concentrate on security issues higher in the stack. The less you have to worry about, the more you can focus on what's left. This new ability to focus is good, but it will also put many of you into unfamiliar territory. We understand things like OS patching and network controls because we've been doing it. But how many of us understand application security?

Let's discuss how security changes with serverless.

The Security Responsibilities You Don't Have

If you're familiar with the **AWS shared responsibility model for security**, then you're familiar with ceding responsibility, and control, over different layers of security to your public cloud provider. For example, physical security and hypervisor patching is the responsibility of the public cloud provider.

That's a good thing! Even in the most sophisticated companies, there is usually more work than a security team can handle. (That assumes you even have a security team, which many organizations simply roll into the operations position already.) Cloud providers like AWS have a large, dedicated security team.

With serverless, the cloud provider assumes even more responsibility. Gone is the time spent worrying about host OS patching. Take the recent **Meltdown and Spectre attacks**. **AWS Lambda required no customer intervention**. AWS assumed responsibility for testing and rolling out patches.

Compare that with stories of organizations tracking patch announcements, testing, rolling out patches (and rolling back in some cases), and the overhead incurred from the disclosure. A month after disclosure, **just one third of companies had patched only 25 percent of their hosts**. Moving more work to the large, dedicated security teams who support the public cloud providers will enhance the security posture of most organizations.

The shared responsibility model lets you spend less time on those areas and more time focusing on other security layers.

The Security Responsibilities You Do Have

So what are the responsibilities of an operations person with regard to the security of serverless systems?

Preparing for the Unplanned

Your first responsibility with security is more or less a reliability responsibility. Any infrastructure change may result in system reliability issues, and the same goes for security updates.

AWS makes regular changes to their infrastructure; some announced, but most not. How do you handle those changes today? If you're coming from the on-premises world, remember, you can't ask AWS to stop patching because your system broke.

To start, you have alerts for system instability and performance degradation. When AWS announces security updates to the EC2 hypervisor layer, you watch dashboards more closely during the process.

> **"Team, someone has disclosed a vulnerability with a website and logo. Keep a closer eye on alerts and the reliability of your services. AWS has released a bulletin that they will be rolling out patches to their infrastructure."**

If a system is critical to your organization, then you should be investigating a multi-region architecture and the ability to fail over or redirect more traffic to a different region. None of this changes with serverless, but it's worth reiterating that if you're using a public cloud provider, then the impact of security changes and updates you don't control is already something you deal with and handle.

Cloud Infrastructure Access and Auditing

As for securing serverless infrastructure and systems, let's discuss a few areas. This isn't an exhaustive list of what to secure, but it provides a good overview of where to start.

Start with the most basic of low-hanging fruit. Are there S3 buckets that should not be open? If you're going serverless you'll find yourself **hosting frontend applications and static assets in S3**, which means determining what buckets should and should not be open.

> **"Glad I caught that open S3 bucket or else we'd be this week's winner of the Last Week in AWS S3 Bucket Negligence Award."**

Similar to S3 bucket permissions, spend more time auditing IAM roles and policies for least privileged access. It can be tempting for a developer to write IAM roles with widely permissive access. But if a Lambda function only needs to read from DynamoDB, then make sure it doesn't have permission to write to it.

Make sure the function can only read from the one table it's intended to read from, too. This may sound obvious, but the current state of cloud security and mishaps that occur make this all worth repeating.

Understand that not everyone has the same level of knowledge or sophistication when it comes to permissions and access. A developer may not know the difference between DynamoDB GetItem and BatchGetItem operations, but they know they can write ***dynamodb:**** and be unblocked. The developer may not know **how to get the DynamoDB table name from their CloudFormation stack**, but they know they can use a wildcard and get unblocked. As a member of the team, you should be finding these issues, correcting them, and educating your team on best practices.

> **"I see you have a system that writes to a DynamoDB table. I went over the IAM role for it and it's too wide. Do you have time for me to show you how I fixed it and what you can do in the future?"**

Finally, ensure that event and action auditing services, for example AWS CloudTrail, are set up. That will give you visibility into what's occurring in the environment. You want to know about repeated AccessDenied failures and activity in an unexpected AWS region.

> **"I don't know how, but someone is mining bitcoin in our account over in ap-southeast-2 . . ."**

There's a variety of tools and products already available to help you audit your cloud infrastructure for best practices.

Application Security

This area will probably be the newest to most of us. It absolutely is for me. Topics like runtime application security, static analysis, and other AppSec areas aren't ones I've spent much time with. But there are still several areas of application security we can pick up quickly.

Start with application dependencies. You're so busy patching the host OS today; how much time does that leave you to ensure your application's third-party dependencies aren't vulnerable? With a public cloud provider patching the application runtime environment, many organizations can finally move up the stack. Just because you didn't write it doesn't mean you don't have to secure it.

Next, who or what is talking to your API endpoints? If you're using AWS API Gateway, you will probably have exposed public HTTP endpoints. (You can put them into a VPC, but then you've introduced the complexity of VPC management and increased latency during AWS Lambda cold-starts.) You'll need to ensure that functions invoked by API Gateway have

proper authentication and authorization controls. You as the operations person will need to ensure that your chosen auth provider, e.g. **AWS Cognito** or an identity SaaS provider, are properly integrated into API Gateway and function code.

Finally, another important attack vector will be "event injection." You're probably familiar with SQL injection attacks, a form of data injection, where unsanitized inputs lead to malicious or unexpected database activity.

In an event-driven architecture, every function will be triggered by some sort of event input. You can't assume that the event's data is safe. You might believe that an earlier function in the processing chain has already validated its input and the data passed to your function is safe. You'd be wrong.

First, you're coupling your function to the behavior of another upstream function, which can lead to issues when the upstream function changes. Second, you're also assuming that there is only one pathway to your function, which isn't definitive. This old issue of data injection is still present and the threat surface has arguably increased due to the proliferation of small-event-triggered functions in your environment.

If you're new to application security, as I am, then have a look at the **Serverless Security Top 10**. This gives a good primer on the threat and attack vectors of serverless architecture.

The Future Role Of Security

I want to close by pointing out, security is just like operations in that it can be a responsibility, role, team, or some permutation of those. The security space will also have to redefine their role under serverless.

I've made the argument previously that serverless infrastructure results in operations teams not making enough sense anymore. A monolithic operations team won't be as effective because they'll be gradually minimized in the service delivery process. Ops teams should be dissolved and their members redistributed to cross-functional feature development teams. The same issues of being bypassed in the service delivery process will happen to security professionals. Arguable, it already does happen to them in many organizations.

I'm not really not sure where security professionals belong in the future. But this is an important topic for that community to address as I've been addressing the role of ops.

Chapter 10:

Cost, Revenue, & FinDev

"LAMBDA COMPUTE IS HIGHLY ELASTIC, BUT YOUR WALLET IS NOT."

While many engineers may not be that interested in cloud costs, the organizations that pay our salaries are. Cost is ever present in their mind. The consumption-based — as opposed to provisioned capacity-based — cost of serverless creates new and unique challenges for controlling cloud spend. As your applications usage grows, your costs grow directly with it. But, if you can track costs and revenue down to the system or function level (a practice called FinDev), you can help your organization save money, and even grow.

Additionally, as we progress up the application stack, and therefore up the value chain, those of us in product or SaaS companies will look at revenue numbers for our team's services. We typically treat cost alone as an important metric, when really it's the relationship between cost and revenue that's more important. An expensive system cost can be offset or made irrelevant by revenue generation. As members of product pods tasked with solving business problems, we'll be responsible for whether or not what we've delivered has performed.

Cost Isn't the Only Consideration

To start, focusing on cost alone isn't helpful. It leads to poor decision-making because it's not the entirety of your financial picture. Costs exist within the context of a budget and revenue.

If your cost optimization results in only fractions of a percent in savings on your budget, you have to ask if the work was worthwhile. Saving $100 a month matters when your budget is in the hundreds or low thousands per month. But it doesn't really matter when your budget is in the hundreds of thousands per month.

You have to spend money to make money, as well. Your revenue generation from a product or feature could also make your cost optimization work largely irrelevant. If your product is sold on a razor-thin margin, then cost efficiency is probably going to count. But if it's a high-margin product, then you're afforded a degree of latitude in cost inefficiency. That means you're going to have to understand to some degree how your organization works and how it generates revenue.

Stop looking at your work as just a cost, but as work that generates part of your organization's revenue, too! After all, how long does it take to start realizing cost savings once you factor in developer productivity?

Serverless Financial Work

Serverless uses a consumption model for billing, where you only pay for what you use, and a change in cost month over month may or may not matter. A bill going from $4 per month to $6 per month doesn't really matter. A bill going from $40k per month to $60k per month will probably matter. You can begin to see the added billing complexity that serverless introduces. Cost should become a first-class system metric. We should be tracking system cost fluctuations over time. Most likely we'll be tracking system costs correlated with deploys so when costs do jump we'll have the context to understand where we need to be looking.

Let's start with the immediate financials challenges of serverless. It's consumption based pricing and resulting variability creates some new and interesting challenges.

Calculating Cost

To start, it can be difficult to determine what is a suitable cost to run serverless, as opposed to in a container or on an EC2 instance. There's a lot of conflicting cost information. You'll find everything from serverless being significantly less expensive to serverless being significantly more expensive. The fact is, there's no simple answer. But that means there's useful work for you to do.

Determining the cost benefit of moving a few cron jobs, slack bots, and low-use automation services is easy. But once you try and figure out the cost of a highly used complex system, the task becomes harder. If you're attempting to do a cost analysis, pay attention to these three things:

- **Compute scale**
- **Operational inefficiency**
- **Ancillary services**

When it comes to compute, start by ensuring you're comparing apples to apples as well as you can. That means first calculating what the cost of a serverless service would be today as well as, say, a year from now based on growth estimates. Likewise, make sure you're using EC2 instance sizes you would actually use in production, as well as the number of instances required.

Next, account for operational inefficiency. An EC2-based service may be oversized for a variety of reasons. You may need only one instance for your service, but you probably have more for redundancy. You may have more than you need because of traffic bursts, or because someone has not scaled down the service from a previous necessary high.

Finally, think about ancillary services on each host. How much do your logging, metrics, and security SaaS providers cost per month. All these will give you a more realistic approach to cost.

> **"This will cost you more than $5 per month on EC2 because you cannot run this on a single t2.nano with no metrics, monitoring, or logging in production."**

The major cloud providers release case studies touting the cost savings of serverless, and I've had my own discussions with organizations. I've seen everything from "serverless will save you 80 percent" to "serverless costs 2X as much". Both could be true statements, but the devil is in the details. Does the organization that saved so much money have a similar architecture to yours? Is the 2X cost a paper calculation or one supported by accounting?

The organization that gave me the 2X calculation followed up with operational inefficiency and ancillary service costs eating up most of their savings, to the point they considered serverless and EC2 roughly even. For that reason, they require a cost analysis and convincing argument for a decision not to go serverless.

Preventing Waste

Next, let's talk about how to keep from wasting money. Being pay-per-use, reliability issues have a direct impact on cost. Bugs cost money. Let's look at how with a few examples.

Recursive functions, where a function invokes another instance of itself at the end, can be a valid design choice for certain problems. For example, attempting to parse a large file may require invoking another instance to continue parsing data from where the previous invocation left off. But anytime you see one, you should ensure that the loop will exit, or you may end up with some surprises in your bill. (That has happened to me. Ouch.)

Lambda has **built-in retry behavior**, as well. Retries are important not just for building a successful serverless application, but for a successful cloud application in general. But each retry costs you money.

You might look at your metrics and see a function has a regular rate of failed invocations. You know from other system metrics, however, that the function eventually processes events successfully, and the system as a whole is just fine. While the system works fine, those errors are an unnecessary cost. Do you adjust your retry rate or logic to save money? Before you start refactoring, take some time to calculate potential savings from a refactor over the cost in time and effort.

There's also potential architectural cost waste that can occur. If you're familiar with building microservices and standardizing HTTP as a communication transport, then your first inclination may be to replicate that using API Gateway and Lambda. But API Gateway can become expensive. Does it make more sense to switch to SNS or a Lambda fan-out pattern (where one Lambda directly invokes another Lambda function) for inter-service communication? There's no easy answer to that question, but someone will have to answer it as your team designs services.

Application Cost Monitoring

We should be monitoring system cost throughout the month. Is the system costing you the expected amount to run? If not, why? Is it because of inefficiency or is it a reflection of growth?

The ability to measure cost down to the function level — and potentially the feature level — is something I like to call application cost monitoring. To start, enable **cost allocation tags** in your AWS account. Then you can easily track cost-per-function invocation and overall system cost over time. Overlay feature deploy events with that data and you can understand system cost changes at a much finer level.

Picture how you scale a standard microservice running on a host or hosts. Your options are to scale instances either horizontally or vertically. When scaling horizontally you're adjusting the number of instances that can service requests. With vertical scaling you're adjusting the size of your instances, typically in relation to CPU and/or memory, so that an instance has enough resources to service a determined rate of requests. When these system falls outside of spec in terms of performance you right-size the system by scaling in the appropriate direction.

Each feature or change to a microservice's codebase usually has only a minimal effect on cost. (I say usually because some people like to ship giant PRs and watch their services be ground into dust under real load, requiring frantic scaling.) Your additional new feature does not have a direct effect on cost unless it results in a change in scaling the service vertically or horizontally. It's not individual changes, but the aggregate of them over time that affects changes.

But with serverless systems it's different. System changes are tightly coupled with cost. Make an AWS Lambda function run slightly slower and you could be looking at a reasonable jump in cost. Based on AWS's pricing of Lambda where they round up to the nearest 100ms, a function that goes from executing with an average duration of 195ms costs roughly 45% more if it starts averaging 205ms. Additionally, increasing function RAM raises cost . . . But that might result in shorter function invocation durations so you end up saving money. And these calculations don't even take into account the situations where a system is reconfigured and new AWS resources such as SNS topics, SQS queues, and Kinesis streams are added or removed.

As you can see, cost needs to become a first-class system metric with serverless. We also need tools to help us model cost changes to our serverless systems. All of this is because

if cost monitoring and projection informs us about money already or about to be spent, the next topic will bring together money spent, money allocated, and money generated.

The Future Into FinDev

Application cost monitoring helps yield a new practice popularized by **Simon Wardley called FinDev**. We're going to be including more than just cost and budgets into our engineering decisions. If we can track cost down to the system or even function level, can we take this a step further and track revenue generation down to that level? If we can, then we can include revenue, either existing or projected revenue, in addition to cost and budgets to form a fuller financial picture of engineering effort and productivity.

What Is FinDev?

This requires bridging finance, PM (either product or project management), and engineering (both at the practitioner and leadership level) at a minimum. We want to track cash flow through the organization so engineering efforts can be directed toward making decisions with the greatest business impact. This efficiency has the potential to become a competitive advantage over those who are unable to prioritize their engineering time toward providing the most value.

It starts with tracking revenue into an organization and mapping it back to engineering work and running production systems. If there's a change in revenue month over month then why? Has revenue picked up because a new feature or service has gone to production? Has it decreased because a service is failing to perform adequately? With this relationship

established, we can assign monetary values to systems. And now we can look at our technical systems with a much fuller financial picture surrounding them.

We can also establish a feedback loop in our organization, now. Revenue data and system value should then be made available to engineering leadership and PMs in order to prioritize work properly. You have two serverless systems exhibiting issues, which do you prioritize? If there's a significant difference in value between systems, prioritize the more valuable one. If you're evaluating enhancements, the question becomes even more interesting. Does your team prioritize generating more money out of an existing service, or does it prioritize enhancing an underperforming service?

Last but not least, engineering and PMs should close the loop by measuring success. Has engineering returned a system to its previous financially performant state? Has work decreased or increased revenue? Has a change increased revenue but cost has eroded profit margin? These are all interesting questions to spur the next cycle of engineering work.

Keep in mind, for many of the preceding questions there is no right answer. The data doesn't make decisions for you. It helps you to make more informed decisions.

Applying FinDev Ideas Today

It should be noted that assigning value to systems and prioritizing work is not an entirely new concept. Many organizations already assign dollar values to systems and prioritize work based on that value. But the more granular we can get with assigning value the more granular we can get within an organization's hierarchy for prioritizing work.

Already, good product companies attempt to measure the success of what they deliver through metrics like user adoption and retention. Their next step is to understand revenue generation down to the technical level.

But even in non-product companies there will be room to apply these principles. For example, IT delivers a service that reduces friction in the sales process. Does that service lead to increased revenue for your organization? Imagine being able to answer that question through analytics before yet understanding the reason why. Then picture how absurd it might be to deprecate a revenue-generating service using cost alone as a justification.

More To Define

There's still a lot of room to figure out what the new processes and their implementation around FinDev will be. Those processes will be highly dependent on your organization and business. In addition, how to tie revenue generated down to the system or function level is still a largely unanswered question. There's nothing out there in the market attempting to do that.

The process and practices for combining finance with engineering is still a growing and evolving area. Keep in mind, this is all closely aligned with why we adopted DevOps; to become more efficient and provide more value to our organizations. FinDev is just an extension of that and a new space serverless better opens up.

We're Not There Just Yet

Admittedly, right now you're talking about small dollar amounts, and the cost saving over EC2 may already be enough not to care at the feature level. In the future, however, as we become accustomed to the low costs of serverless, we will care more. Similarly, many organizations still invest heavily in undifferentiated engineering. But as companies that focus more heavily on their core business logic through serverless begin to excel, we'll see more organizations become interested in achieving that level of efficiency.

Chapter 11:
Epilogue

I wrote this ebook for the same reasons I started ServerlessOps. When I first started building serverless applications I was immediately struck by how much I liked it and how it was something truly something different in technology like we hadn't seen since public cloud appeared.

However I soon became apprehensive. I knew this would have an impact on my profession and it brought up a familiar feeling. I was not unaffected in my career by the progress of technology impacting me negatively. It's an experience I do not wish wish to repeat.

On spending more time building and working with serverless applications, and the success I've had, I don't fear serverless displacing me anytime in the future. In fact, I've found it a way to enhance my operations role and responsibilities. And the time spent focusing less on infrastructure and more on the code to solve my problems has leveled-up my coding skills in ways I never expected to achieve.

On finishing this ebook, I'm far less worried about the place of operations as a whole while organizations adopt serverless. I've shown that there is a place for operations people as there is still much engineering work to be done by us.

But there's still work to be done educating and preparing operations engineers and their organizations to make them successful with serverless. It starts with a cultural and individual mindset shift that values the delivery of results most of all. Achieving these results requires adopting an engineering cadence that iteratively delivers work, obtains feedback and measures results, and is ready to course correct or pivot when needed. On the technical side, we need to level up the cloud architecture and coding skills of many in the operations field.

Effort combined with guidance and training makes succeeding not just doable, but something we can even excel at. Good luck!

About ServerlessOps

Started in 2018, **ServerlessOps** provides advisory and technical services around DevOps transformation and AWS serverless cloud infrastructure. Our value delivery focused approach starts by working with leadership in identifying needs, determining goals, and prioritizing work. Taking advantage of the velocity serverless provides, we work with engineering to ensure not just delivery but also the meeting of established goals.

Our range of services include:

DevOps Cultural Transformation

AWS Serverless Migration & Adoption

AWS serverless training

Startup Cloud Operations

See the **ServerlessOps services page to learn more**.

Learn more