97 Things Every Cloud Engineer Should Know

Edited by Emily Freeman & Nathen Harvey

O'REILLY®

97 Things Every Cloud Engineer Should Know

If you create, manage, operate, or configure systems running in the cloud, you're a cloud engineer—even if you work as a system administrator, software developer, data scientist, or site reliability engineer. With this book, professionals from around the world provide valuable insight into today's cloud engineering role.

These concise articles explore the entire cloud computing experience, including fundamentals, architecture, and migration. You'll delve into security and compliance, operations and reliability, and software development. And examine networking, organizational culture, and more. You're sure to find 1, 2, or 97 things that inspire you to dig deeper and expand your own career.

Emily Freeman is a technologist and a storyteller who helps engineering teams improve their velocity. As the author of *DevOps for Dummies*, she believes the biggest challenges facing developers aren't technical but human. Her mission is to transform technology organizations by creating company cultures in which diverse, collaborative teams can thrive. Emily is a principal cloud advocate at Microsoft.

Nathen Harvey is a cloud developer advocate at Google, helping the community understand and apply DevOps and SRE practices in the cloud. He's part of the devopsdays global organizing committee and was a technical reviewer for the *Accelerate State of DevOps Report*. Nathen formerly led the Chef community, cohosted the *Food Fight Show*, and managed operations and infrastructure for a diverse range of web applications.

Three Keys to Making the Right Multicloud Decisions
—Brendan O'Leary

Serverless Bad Practices
—Manasés Jesús Galindo Bello

Failing a Cloud Migration
—Lee Atchison

Treat Your Cloud Environment as If It Were On Premises
—Iyana Garry

What Is Toil, & Why Are SREs Obsessed with It?
—Zachary Nickens

Lean QA: The QA Evolving in the DevOps World
—Theresa Neate

How Economies of Scale Work in the Cloud
—Jon Moore

CLOUD

9 781098 108472

Twitter: @oreillymedia
facebook.com/oreilly

Build smarter.
Ship faster.

To make the most of the cloud, IT needs to approach applications in new ways. Cloud-native development means packaging with containers, adopting modern architectures, and using agile techniques.

Red Hat can help you arrange your people, processes, and technologies to build cloud-ready apps. See how at **https://red.ht/cloud-native-development**

97 Things Every Cloud Engineer Should Know

Collective Wisdom from the Experts

Emily Freeman and Nathen Harvey

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

97 Things Every Cloud Engineer Should Know
by Emily Freeman and Nathen Harvey

Printed in the United States of America.

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

Acquisitions Editor: Jennifer Pollock
Development Editor: Sarah Grey
Production Editor: Christopher Faucher
Copyeditor: Sharon Wilkey
Proofreader: Rachel Head

Indexer: Potomac Indexing, LLC
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: O'Reilly Media, Inc.

December 2020: First Edition

Revision History for the First Edition
2020-12-04: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492076735* for release details.

978-1-098-10847-2

[LSI]

Black Lives Matter.

Table of Contents

Part II. Architecture

Part VI. Software Development

Preface

Ideas about cloud computing have been around since at least the 1960s. Our modern understanding of the cloud can be traced back to about 2006, when Amazon first launched Elastic Compute Cloud (EC2). The rise and adoption of cloud technologies has changed the shape of our industry and our global society. The cloud has made getting started less expensive and growing to global scale feasible, and is helping turn every organization into a technology organization—or at least an organization that uses technology as a strategic enabler of delivering value.

A *cloud engineer* is, broadly defined, someone who creates, manages, operates, or configures systems running in the cloud. This could be a system administrator responsible for building base images, a software developer responsible for writing applications, a data scientist building machine learning models, a site reliability engineer responding to pages when things go awry, and more. In some organizations, all of those functions are handled by one single human; in others, hundreds of people may be in each one of those roles.

This book is a collection of articles from a diverse set of professional cloud engineers. We have authors from around the world. Some are early in their cloud journeys, and others have decades of experience. Each and every author brings their own perspective and experience to the article that they've shared as part of this book. Our intent is to help you find one, two, or maybe even ninety-seven things that inspire you to dig deeper and expand your own career. Just as the cloud has many facets, this book has many types of articles for you to check out. Start with some cloud fundamentals, and then read more about software development approaches in the cloud. Or start with a couple of articles about how to improve your organization, then dig into new approaches to operations and reliability. It really is up to you!

This book was written in 2020, a year marked by a global pandemic, an amplification of and broader awakening to the injustices of systemic racism, and many other changes that will have an effect on generations to come. The events of this year have touched every one of us on a personal level. Companies and organizations are not immune to these events either: 2020 saw some companies experience explosive growth, while others had to face their swift demise. The cloud has played a role in all of these things too—whether providing new ways for us to connect while remaining socially distant, rapidly spreading information and misinformation, or providing scientists the technology required for testing, tracing, and learning more about a pernicious virus.

We would like to thank the authors of each article. They have generously shared their insights and knowledge in an effort to inform and inspire as you continue your own journey as a cloud engineer. Use this book to spark a conversation among cloud engineers, connect on a human level, and learn from one another.

Enjoy the book!

— Nathen Harvey and Emily Freeman

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/97-things-cloud-engineers*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

Visit *http://oreilly.com* for news and information about our books and courses.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

PART I

Fundamentals

What Is the Cloud?

Nathen Harvey

Developer Advocate at Google

Before you get too deep into the articles in this book, let's establish a common understanding of the cloud.

Wikipedia says that *cloud computing* is "the on-demand availability of computer system resources, especially data storage (cloud storage) and computing power, without direct active management by the user." The term is generally used to describe datacenters available to many users over the internet.

At its most basic level, the cloud is essentially a datacenter that you access over the internet. However, viewing the cloud as "someone else's datacenter" does not really allow you or your team to take full advantage of all that the cloud has to offer.

The National Institute of Standards and Technology (NIST) defines the cloud model in "SP 800-145: The NIST Definition of Cloud Computing". This publication covers the essential characteristics of cloud computing. It's a quick read and well worth your time.

NIST outlines five essential characteristics of the cloud model:

On-demand self-service
: Cloud resources of all varieties—compute, storage, databases, container orchestration platforms, machine learning, and more—are available at the click of a button or by calling an API. As a cloud engineer, you should not need to call someone, open a ticket, or send an email to provision, access, and configure resources in the cloud.

Broad network access
: As a cloud engineer, you should be able to utilize the self-service capabilities of the cloud wherever you are. A cloud provides authorized users access to resources over a network that you can connect to using a variety of devices and interfaces. You may be able to restart a service from

your mobile phone, ask your virtual assistant to provision a new test environment, or view monitors and logs from your laptop.

Resource pooling
: Cloud providers pool resources and make them available to multiple customers—with security and other protections in place, of course! Practically speaking, a cloud engineer does not need to know the physical location of the CPU in the datacenter. Pooling also provides higher levels of abstraction. A cloud engineer may specify the compute and memory requirements for an application, but not which physical machines provide the computing resources. Likewise, a cloud engineer may specify a region where data should be stored but would not have any say over which datacenter rack houses the primary database.

Rapid elasticity
: A cloud engineer should not need to worry about the physical capacity of a particular datacenter. Resources in the cloud are designed to scale up to meet demand. Likewise, when demand for a service decreases, cloud resources are designed to contract. Remember, elasticity goes both ways: scale up and scale down. This scaling may happen at the request of a cloud engineer, made via a user interface or API call, though in many instances it will happen automatically with no human intervention.

Measured service
: Consumption of cloud resources is measured and is usually one component of the cost. One of the promises of the cloud is that you pay for what you use, and no more. Having visibility into how much of each type of resource every service is using gives you visibility into your costs that is typically not feasible in a traditional datacenter.

NIST's definition goes beyond these five characteristics of cloud computing to define service models like infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). The article also describes various deployment models, including private and public.

Keep these five characteristics in mind as you explore the cloud. Use them to help evaluate whether you are taking advantage of the cloud or simply treating it as someone else's datacenter.

Why the Cloud?

Nathen Harvey

Developer Advocate at Google

In the early 2000s, I worked in the IT department of a publicly traded software company. My team was not responsible for building the software that our company delivered to customers, but we were responsible for our customer and partner extranet, building a software delivery center that would allow our customers to download the software instead of waiting for compact discs, our internal sales operations tools, and more. We were using technology to enable the business. All of our systems ran in datacenters that we managed. Expanding capacity for our systems required procuring, installing, and configuring new hardware. This cycle could take up to 18 months.

In 2007 that company was acquired, and IT fell under the CIO of the new organization. That person had one primary objective: cut costs. The CIO met with our team and encouraged us to immediately stop working on anything that cost money. We protested: our work supports and enables the business to be more efficient, keeps our customers happy, and leads to more revenue, we argued. But this was no concern of the CIO, whose objective was clear, focused, and dispassionate: keep costs down.

By 2008, I'd left that company and joined my first start-up. We were a small, scrappy team with a singular focus: launch. This was my first exposure to the power of the cloud. A procure-and-provision process that used to take 18 months was now completed in minutes. The cloud has been a fundamental enabler of my career for over a decade now, and I've picked up a few lessons along the way.

Understand the Role of Technology

Yes, it is true that every company is now or is becoming a technology company. It is important to listen to the words and watch the actions of leaders in your organization. Technology can be a path to cost savings, or it can be a key enabler and amplifier of the business. Sure, it can be both at the same time, but your leaders are likely more focused on one of these two outcomes. Pay attention and let their goals help drive the work that you do, or even

where you work. Misaligned incentives can lead to friction, burnout, and unsatisfied customers.

Automate the Cloud

In my preceding article, "What Is the Cloud?" I shared NIST's list of capabilities that define the cloud. It is easy to fall into the trap of not utilizing the cloud properly. Being able to provision resources at the click of a button or the call of an API is just the beginning. How long does it take to go from provisioned to useful? Invest time in learning how to automate and compress that cycle. Doing so opens up a multitude of new ways to manage infrastructure and applications.

Measure Progress

How do you know whether the cloud is working for you? When migrating from a traditional datacenter environment, you will feel and see many immediate improvements. But what if the applications you are responsible for were born in the cloud? One thing is certain: there is always room for improvement. I recommend starting with high-level measures for each team or application and allowing improvements across those metrics to guide your team's investment in improvement. The four keys identified by the DORA research led by Dr. Nicole Forsgren are a great place to start:

- Lead time
- Deploy frequency
- Time to restore
- Change fail percentage

Getting Started > Getting Finished

Aligning incentives, building up your team's automation capabilities, and measuring progress takes time and energy. Matching the success of the best in the industry can seem daunting, or even unachievable. The truth is that you must take an iterative approach. Remember, getting started with improvements is more important than getting finished with them.

Three Keys to Making the Right Multicloud Decisions

Brendan O'Leary

Senior Developer Evangelist at GitLab

In recent years, there has been a lot of discussion about the possibility of multicloud and hybrid-cloud environments. Many business and technology leaders have been concerned about vendor *lock-in*, or an inability to leverage the best features of multiple hyperclouds. In regulated industries, there can still be a hesitancy to move "everything" to the cloud, and many want to keep some workloads within their physical datacenters.

The reality in the enterprise is that multicloud and hybrid-cloud are already here. A 2019 State of the Cloud report found that 84% of organizations are already using multiple clouds. On average, they use more than four clouds. At the same time, we know that software excellence is the new operational excellence. "Software has eaten the world," and our competitiveness depends on our ability to deliver better products faster.

Based on those realities, the question isn't whether you will be a multicloud or hybrid-cloud company. The question is, are you ready to be better at it than your competition? If we accept that a multicloud strategy is required, we need to systemize our thinking. There are three key enablers here to consider: workload portability, the ability to negotiate with suppliers, and the ability to select the best tool for a given job. The cloud promises to remove undifferentiated work from our teams. To realize that potential, we must have a measured approach.

The most critical enabler is workload portability. No matter what environment a team is deploying to, we must demand the same level of compliance, testing, and ease of use. Thus, creating a complete DevOps platform that is cloud-agnostic allows developers to create value without overthinking about where the code deploys.

In considering both the platform your developers will use and how to make the right multicloud decisions, there are three keys: visibility, efficiency, and governance.

Visibility means having information where it matters most, a trusted single source of truth, and the ability to measure and improve. Whenever considering a multitool approach—whether it is a platform for internal use or the external deployment of your applications—visibility is crucial. For a DevOps platform, you want real-time visibility across the entire DevOps life cycle. For your user-facing products, observability and the ability to correlate production events across providers will be critical for understanding the system.

Efficiency may seem straightforward at first, but there are multiple facets to consider. We must always be sure we are efficient for the right group. If a tools team is selecting tools, the bias may be to optimize for that team's efficiency. But if a selection here saves the tools team, which has 10 people, an hour a week but costs 1,000 developers even a few extra minutes a month, a negative impact on efficiency results. Your platform of choice must allow development, QA, security, and operations teams to be part of a single conversation throughout the life cycle.

Finally, *governance* of the process is essential regardless of industry. However, it has been shown that working governance into the day-to-day processes that teams use allows them to move quicker than a legacy end-of-cycle process. Embedded automated security, code quality, vulnerability management, and policy enforcement practices enable your teams to ship code with confidence. Regardless of where the deployment happens, tightly control how code is deployed and eliminate guesswork. Incrementally roll out changes to reduce impact, and ensure that user authentication and authorization are enforceable and consistent.

These capabilities will help you operate with confidence across the multicloud and hybrid-cloud landscape.

Use Managed Services—Please

Dan Moore

Principal at Moore Consulting

Use managed services. If there was one piece of advice I could shout from the mountains to all cloud engineers, this would be it.

Operations, especially operations at scale, are a hard problem. Edge cases become commonplace. Failure is rampant. Automation and standardization are crucial. People with experience running software and hardware at this scale tend to be rare and expensive. The knowledge they've acquired through making mistakes and learning from different situations is hard-won.

When you use a managed service from one of the major cloud vendors, you're getting access to all the wisdom of their teams and the power of their automation and systems, for the low price of their software.

A managed service is a service like Amazon Relational Database Service (RDS), Google Cloud SQL, or Microsoft Azure SQL Database. With all three of these services, you're getting best-of-breed configuration and management for a relational database system. Configuration is needed on your part, but hard or tedious tasks like setting up replication or backups can be done quickly and easily (take this from someone who fed and cared for a MySQL replication system for years). Depending on your cloud vendor and needs, you can get managed services for key components of modern software systems, including these:

- File storage
- Object caches
- Message queues
- Stream processing software
- Extract, transform, load (ETL) tools

(Note that these are all components of your application, and will still require developer time to thread together.)

There are three important reasons to use a managed service:

- It's going to be operated well. The expertise that the cloud providers can provide and the automation they can afford to implement will likely surpass your own capabilities, especially across multiple services.
- It's going to be cheaper. Especially when you consider employee costs. The most expensive Amazon RDS instance costs approximately $100,000 per year (full price). It's not an apples-to-apples comparison, but in many countries you can't get a database architect for that salary.
- It's going to be faster for development. Developers can focus on connecting these pieces of infrastructure rather than learning how to set them up and run them.

A managed service doesn't work for everyone, though. If you need to be able to tweak every setting, a managed service won't let you. You may have stringent performance or security requirements that a managed service can't meet. You may also start out with a managed service and grow out of it. (Congrats!)

Another important consideration is lock-in. Some managed services are compatible with alternatives (Kubernetes services are a good example). If that is the case, you can move clouds. Others are proprietary and will require substantial reworking of your application if you need to migrate.

If you are working in the cloud and need a building block for your application, like a relational database or a message queue, start with a managed service (and self-host if it doesn't meet your needs). Leverage the operational excellence of the cloud vendors, and you'll be able to build more, faster.

Cloud for Good Should Be Your Next Project

Delali Dzirasa

Founder and CEO of Fearless

I don't know about you, but being stuck on an endless phone call with an automated system drives me crazy. Usually, I'm trying to solve a simple problem or access a service, but it can feel like I'm trapped in an endless loop of pressing 1 or 2 to *respond* to questions rather than getting answers myself.

Why is it that I can press a few buttons on my phone and, in minutes, have a car or food waiting for me, but I can't figure out how to easily pay my water bill online?

The cloud powers everything these days. Or, at least, it powers everything that you enjoy using, usually because the tech it supports makes your day-to-day life easier in some way. But, oddly, the real-world problems that most people care about—problems around things like education, healthcare, and food security—don't get nearly enough attention.

For groups dealing with meaningful social issues, tech can be the last thing on their list. There often just isn't enough focus, energy, or resources available to make meaningful tech improvements.

On a macro level, this speaks to a real gap in the market: there simply aren't enough digital services firms focused on helping to support civic tech or *cloud for good*. Fearless, the company I founded in 2009, has a mission of building software with a soul. We take on only projects that empower users and change lives.

One of the places we're working to build software with a soul is the Centers for Medicare & Medicaid Services (CMS), where we're helping the agency modernize its technology. When programs are inefficient, recipients don't receive the best care, costs are high, and, at the end of the day, it's American taxpayers who pay the price. Improving these technologies makes the healthcare system work better for everyone.

The ideas that power cloud for good have been around for a long time—longer than the terms *civic tech* or *cloud for good* themselves. For me, cloud for good really came into the forefront of my mind when HealthCare.gov failed. People wanted to help, and a bunch of digital services firms came in to help.

I've heard from a lot of people who are saying, "How can I use my tech powers for good? I want to work on projects that solve problems." I believe this speaks to the larger movement of people looking for meaning in the work that they do and wanting humanity and our world to be better.

Get involved with local meetups, especially ones centered around solving problems in the cloud-for-good space in your community. Code for America brigades are a good place to start if you're looking for outlets that you can work with.

If there's a nonprofit organization in your area that you would like to support, donate your time to help build software. Think of the needs of civic tech when you're writing code. Open source software allows more people to benefit from and use software solutions. By building open source, you enable others to leverage your tech to support more projects.

In my city, we have Hack Baltimore. The tech movement teams up community advocates, nonprofits, technologists, and city residents to design sustainable solutions for the challenges impacting Baltimore. Find an organization like Hack Baltimore in your community, or start one.

The cloud isn't inherently good or bad; it's all based on the intent of the end users and those of us who wield our "tech powers" to power applications around the world. We can help power amazing social missions that too often get left behind. So while you're off building the cloud, consider using some of that energy to build the cloud for good.

A Cloud Computing Vocabulary

Jonathan Buck

Senior Software Engineer at Amazon Web Services

In any profession, being able to speak and understand the vernacular goes a long way toward feeling comfortable in your role and working effectively with your colleagues. If you're just starting your career as a cloud engineer, you will likely hear these terms throughout your workplace:

Availability
: The amount of time that a service is "live" and functional. This is often expressed in percentage terms. For example, if someone says their service has a yearly availability of 99.99%, that means it will be unavailable for only 52.56 minutes in an entire year.

Durability
: Even for the most reliable devices, any data stored on a computer is ephemeral over a long enough time frame. *Durability* refers to the chance that data will be accidentally lost or corrupted over a given time period. Like availability, it's typically expressed as a percentage value.

Consistency
: *Consistency* refers to the notion that when you write data to a data store, it (or the latest version of it) might not be immediately available. This is because cloud-based data stores are built on distributed systems, and distributed systems are subject to the CAP theorem. (Also known as Brewer's theorem, it holds that there are three things a distributed data store can guarantee—consistency, availability, and partition tolerance—but it can never guarantee all three at the same time.) Different cloud environments and services will handle this differently, but the important thing is to be aware of this nuance in designing cloud-based software applications.

Elasticity

One of the main advantages of the cloud is *elasticity*: the ability to dynamically match hardware or infrastructure with the demands being placed on an application at any given time. Elasticity has benefits in two directions. Sometimes, like during high-traffic periods, you might need more resources; at other times, you might need fewer. Because your resource cost will be proportional to your provisioned resources, elasticity is a means of better matching your costs with the actual load on your application.

Scalability

Scalability is similar to elasticity. Whereas *elasticity* refers to the notion of dynamically increasing or decreasing your resources, *scalability* refers to how your resources are actually augmented. Typically, scalability is decomposed into two concepts: horizontal scaling and vertical scaling. *Horizontal scaling* refers to adding host machines in parallel to meet application demand. *Vertical scaling* refers to adding resources within a given machine (such as adding RAM). These scaling approaches have advantages and disadvantages, and are appropriate in different scenarios. The proper choice depends on your system architecture.

Serverless

Serverless refers to modern technology that allows you to run application code without managing servers, hardware, or infrastructure. Sometimes this capability is also referred to as *function as a service* (FaaS). Many cloud providers offer their own forms of this. These serverless offerings also provide the benefits of high availability and elasticity, concepts discussed previously. Before serverless technologies, deploying software involved managing and maintaining servers, as well as working to make them available and scalable to meet traffic needs. Using serverless compute environments saves you from the burden of managing and maintaining servers in this fashion so you can focus your time and energy on the application code. However, various trade-offs are involved.

Fully managed

In the early days of cloud computing, we typically interacted with basic computing resources that were made available in the cloud: servers, data stores, and databases. While this provided significant advantages, the responsibilities of the software engineer were largely the same as those in on-premises datacenters: managing and maintaining low-level hardware resources. *Fully managed resources* are cloud resources that are offered at a higher level of abstraction. The cloud provider takes responsibility

for some aspects of these resources, rather than the software engineer. The trade-off is that fully managed services are typically more expensive as a result, and often introduce various limitations or restrictions as compared to operating on the more basic resources.

Why Every Engineer Should Be a Cloud Engineer

Michelle Brenner

Senior Software Engineer

I am a lazy engineer. If I have the option to copy, reference, or install a tool to get my job done faster, I say thanks and move on. I started my tech career working in entertainment, where you don't have sprints or quarters to finish tools. Studios need the work done yesterday, because they want to get the best art possible before the release date. As an engineer, I know I can solve any problem, but I've realized that I can have a much greater impact using available tools.

Most of us are not working on groundbreaking technology; we're solving problems for customers. I want to focus on delighting my clients, not fiddling with a problem a thousand engineers have already tackled. It is a common misconception that cloud computing is just servers in someone else's warehouse. While you can get that bare-metal setup, there are so many other features that no single person can know them all. If you can define a problem in a general way, such as *log in using social accounts*, *store information securely*, or *scale service to meet demand*, you can find a cloud tool to do it.

As a backend engineer, I was building APIs and designing databases. Learning cloud technologies meant I could get my own code live faster, more easily, and more reliably. A colleague was not always available to help me improve the deployment pipeline or debug production problems. Learning how to make these changes myself made the whole team more efficient and effective. Expanding my skill set opened doors to career opportunities and made it easier to accomplish personal projects.

Before I got started in cloud computing, I often abandoned personal projects because deploying them seemed so daunting. But after gaining a sufficient understanding of the systems involved, not only could I complete projects, I could even use them to help with seemingly unrelated ones, like hosting a

podcast. I knew that most podcasts don't make money, so I decided that if the costs started to add up, I would not continue. After doing some research, I realized that hosting costs for podcasts had three main features:

- Hosting the public files (audio and images)
- Formatting an XML file for the podcast aggregators
- Tracking episode playbacks

Why would I pay $10 to $15 a month for that when Amazon Simple Storage Service (S3) can host my files for pennies? Hosting myself also meant I did not have to worry about a third party handling my content or data. I set up a public bucket for the audio and image files. Then I wrote up my XML file for the aggregators and put it in the same bucket. To track playbacks, I added logging on those files and analyzed them using Amazon Athena. I learned that I don't have many listeners, but that's OK since my AWS bill is less than $1 a month.

Now that I have you completely convinced to become a cloud engineer, here is some rapid-fire advice I wish I'd gotten before I got started:

- Turn on billing alerts before you do anything else. It's possible you could follow a tutorial, not really knowing what you're doing, and suddenly get a huge bill. Hypothetically.
- Get as many free credits as you can. Your provider is competing with other cloud-hosting providers for your business. Make them earn it.
- The documentation often focuses on features rather than user stories. Independent content creators are great for filling in those gaps. Dev.to is your friend.
- Change only one setting at a time.
- No one understands identity and access management (IAM).

Finally, if I have inspired you to learn and create something new, I'd love to hear about it. Learning new tools will increase your impact, but teaching others how to use them will expand it exponentially.

Managing Up: Engaging with Executives on the Cloud

Reza Salari

Business Information Security Officer

In job descriptions for cloud engineers, you will often see a lot written about the technology stack, programming/scripting languages, and years-of-experience requirements that sometimes exceed how long the technology has actually been available. However, arguably one of the most important job requirements rarely makes the list. It is frequently what makes or breaks your ability to implement a new capability, and it can help you avoid expectations that were never really based in reality anyway. Master it, and you can unlock resources, support, and opportunities for you and your team. Failure, on the other hand, often leads to frustration as your ideas seem to die on the vine or you find yourself saddled with objectives that just can't be met.

We often focus on managing down, but learning to *manage up* and communicate with executives can, and should, be your (not so) secret weapon! Although this skill takes years to cultivate, you can do some practical things now to show them you can talk at their level. Here are my top five tips:

Understand what executives really need for the business.
: Sure, it's exciting when a new technology hits the market, and as technologists we can't wait to put it to good use. However, our focus should be on choosing the right capabilities to answer the unmet needs of the business.

Tell them why what you're proposing will meet their needs, in their language.
: You've found the perfect new capability that will solve a real business problem, but when you tell them about the features, they just can't connect the dots. Drop the jargon, talk about the outcomes, and tell the story of how their experience will improve.

Be a trusted voice in a world of marketing buzzwords and sky-high expectations.

The cloud is one of the great innovations driving technology and businesses forward. There are plenty of real wins and successes to point to. Executives are flooded with sales calls, marketing emails, and anecdotal stories of how some large company built things better, faster, and cheaper, so their company should be able to too. You, as a cloud engineer, have an opportunity to guide them through the noise to set realistic expectations and identify trade-offs. Pragmatism goes a long way!

Know the numbers.

The cloud relies on consumption-based usage for its cost model, whereas legacy on-premises datacenters rely more on making the most of fixed capacity. For example, if you have a grid-computing workload that runs a model for two hours, three times a week, moving that to the cloud may seem to make perfect sense. In a greenfield environment, it certainly would. However, knowing that you have hardware in your datacenter that has already been purchased and has three more years of depreciation left could change which solution you advocate for. Adding financial context to your recommendations demonstrates business acumen and gets to the root of most of their questions.

Know how your executive's performance is measured.

We all are motivated to succeed, and how we measure success as well as how others measure our success guides us. Learn what motivates your executives and what goals they are working toward. Show them you want to be a partner in their success and that of the business.

You have a wealth of insight and knowledge that executives are craving; now go out and tell them all about it in their language!

PART II

Architecture

The Future of Containers: What's Next?

Chris Hickman

VP of Technology at Kelsus

Deciding which technology to use for running your cloud native applications is a question of trade-offs.[1] Virtual machines provide great security and workload isolation but require significant computing resources. Containers offer better performance and resource efficiency but are less secure because they share a single operating system kernel.

What if we didn't have to make these trade-offs? Let's explore two of the most promising technologies that combine the best of virtual machines and containers: microVMs and unikernels.

MicroVMs

MicroVMs are a fresh approach to virtual machines. Rather than being general-purpose and providing all the functionality an operating system *may* require, microVMs specialize for specific use cases.

For example, a cloud native application needs only a few hardware devices, such as for networking and storage. There's no need for devices like full keyboards, mice, and video displays.

By implementing a minimal set of features and emulated devices, microVM hypervisors can be extremely fast with low overhead. Boot times can be measured in milliseconds (as opposed to minutes for traditional virtual machines). Memory overhead can be as little as 5 MB of RAM, making it possible to run thousands of microVMs on a single server.

A big advantage of containers is that they virtualize at the application level, not the server level used by virtual machines. This is a natural fit with our

1 A version of this article was originally published at Upstart.

development life cycle—after all, we build, deploy, and operate applications, not servers.

A better virtual machine by itself doesn't help us much if we have to go back to deploying servers and give up our rich container ecosystem. The goal is to keep working with containers but run them inside their own virtual machine to provide increased security and isolation.

Most microVM implementations integrate with existing container runtimes. Instead of directly launching a container, the microVM-based runtime first launches a microVM and then creates the container inside that microVM. Containers are encapsulated within a virtual machine barrier, without any impact on performance or overhead.

It's like having our cake and eating it too. MicroVMs give us the enhanced security and workload isolation of virtual machines, while preserving the speed, resource efficiency, and rich ecosystem of containers.

Unikernels

Unikernels aim to solve the same problems as microVMs but take a radically different approach.

A *unikernel* is a lightweight, immutable OS compiled to run a single application. During compilation, the application source code is combined with the minimal device drivers and OS libraries necessary to support the application. The result is a machine image that can run without a host operating system.

Unikernels achieve their performance and security benefits by placing severe restrictions on execution. Unikernels can have only a single process. With no other processes running, less surface area exists for security vulnerabilities.

Unikernels also have a single address space model, with no distinction between application and operating system memory spaces. This increases performance by removing the need to context switch between user and kernel address spaces.

However, a big drawback with unikernels is that they are implemented entirely differently than containers. The rich container ecosystem is not interchangeable with unikernels. To adopt unikernels, you will need to pick an entirely new stack, starting with choosing a unikernel implementation. There are many unikernel platforms to choose from, each with its own constraints. For example, to build unikernels with MirageOS, you'll need to develop your applications in the OCaml programming language.

So, What's Next?

If you are using containers, microVMs should be on your road map. MicroVMs integrate with existing container tooling, making adoption rather painless. As microVMs mature, they will become a natural extension of the runtime environment, making containers much more secure.

Unikernels, on the other hand, require an entirely new way of packaging your application. For specific use cases, unikernels may be worth the investment of converting your workflow. But for most applications, containers delivered within a microVM will provide the best option.

Understanding Scalability

Duncan Mackenzie

Developer Lead at Microsoft

A scalable system can handle varying degrees of load (traffic) while maintaining the desired performance. It is possible to have a scalable system that is slow, or a fast site that cannot scale. If you can handle 100 requests per second (RPS), do you know what to do if traffic increases to 1,000 RPS? The cloud is well suited to producing a reliable and scalable system, but only if you plan for it.

Scaling Options

To increase the capacity of a system, you can generally go in two directions. You can increase the size/power of individual servers (scaling up) or you can add more servers to the system (scaling out). In both cases, your system must be capable of taking advantage of these changes.

Scaling Up

Consider a simple system, a website with a dependency on a data store of some kind. Using load testing, you determine that the site gets slower above 100 RPS. That may be fine now, but you want to know your options if the traffic increases or decreases. In the cloud, the simplest path is usually to scale up the server that your site or database is running on. For example, in Azure, you can choose from hundreds of machine sizes, all with different CPU/memory and network capabilities, so spinning up a new machine with different specifications is reasonably easy.

Increasing the size of your server may increase the number of requests you can handle, but it is limited by the ability of your code to take advantage of more RAM, or more CPU cores. Changing the size often reveals that something else in your system (such as your database) is the limiting factor. It is possible to scale the server for your database as well, making higher capacity possible with the same architecture.

It is worth noting that you should also test scaling *down*. If your traffic is only 10 RPS, for example, you could save money by running a smaller machine or database.

Scaling up is limited by the upper bound of how big a single machine can be. That upper limit may cover your foreseeable needs, but it is still an example of a poorly scalable system. Your goal is a system that can be configured to handle *any* level of traffic.

An infinitely scalable system is hard, as you will hit different limits. A reasonable approach is to plan for 10 times your current traffic and accept that work will be needed to go further.

Scaling Out

Scaling out is the path to high scalability and is one of the major benefits of building in the cloud. Increasing the number of machines in a pool as needed, and then reducing it when traffic lowers, is difficult to do in an on-premises situation. In most clouds, adding and removing servers can happen automatically, so that a traffic spike is handled without any intervention. Scaling out also increases reliability, as a system with multiple machines can better tolerate failure.

Unfortunately, not every system is designed to be run on multiple machines. State may be saved on the server; for example, requiring users to hit the same machine on multiple requests. For a database, you will have to plan how data is split or kept in sync.

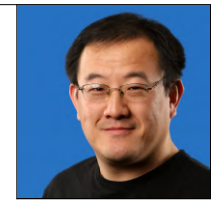
Keep Scalability in Mind, but Don't Overdo It

Consider how your system *could* scale up or down as early as possible, because that decision will guide your architecture. Do you need to know the upper bound? Does everything have to automatically scale? No! Optimizing for high growth too early is unnecessary. Instead, as you gather usage data, continue testing and planning.

Don’t Think of Services, Think of Capabilities

Haishi Bai

Principal Software Architect at Microsoft



Acquiring a continuous power supply is a fundamental capability of a mobile device. Most of us are familiar with sight of people flocking around charging stations at airports (before the pandemic happened). As a matter of fact, because this capability is so critical, we use a mixture of methods to provide a continuous power supply to our precious phones—integrated batteries (in a software sense, in-process libraries), portable power banks (local Docker containers or services), or power plugs (service-oriented architecture, or SOA).

To get your phone working, you don’t really care whether the power comes from a plug or a power bank; you just need the capability to acquire power. Capability-oriented architecture (COA) aims to provide a set of languages and tools for developers and architects to design applications based on *capabilities*, regardless of where and how these capabilities are delivered, which is an operational concern.

COA is especially relevant to edge-computing scenarios. For an edge solution to keep continuous operation, it often needs to switch between service providers when network conditions change. For example, a smart streetlight system sends high-resolution pictures to a cloud-based AI model to detect wheelchairs on a crosswalk and extends the green light as needed. When the network connection degrades, it switches to low-resolution images. And when the network is completely disconnected, it switches to a local model that gives lower detection rates but allows business continuity. This is a sophisticated system with various integration points and decision logic. With COA, all the complexity is abstracted away from developers. All developers need to do is to have a wheelchair detection capability delivered, one way or another.

COA is also relevant to cloud developers for two reasons. First, the cloud and the edge are converging, and compute is becoming ubiquitous. As a cloud developer or architect, you'll face more and more situations that require you to push compute toward the edge. COA equips you with the necessary abstractions to keep your architecture intact while allowing maximum mobility of components. You can imagine your solution as a puddle of quicksilver that spans and flows across the heterogeneous computing plane, across the cloud and the edge. Second, COA offers additional abstractions on top of SOA so that your applications are decoupled from specific service vendors or endpoints. COA introduces a *semantic discovery* concept that allows you to discover capability offerings based on both functional and nonfunctional requirements, including service-level agreements (SLAs), cost, and performance merits. This turns the service world into a consumer market, as consumers are granted more flexibility to switch services, even dynamically, to get the best possible returns on their investments. COA also allows traditional cloud-based services to be pushed toward the edge, onto telecommunications infrastructure or even household devices (such as in-house broadband routers). This will be the foundation of a new breed of distributed cloud without central cores that can't be shut down (think of Skynet in the *Terminator* movies).

With developments in natural language processing, we can imagine COA capability discovery being conducted in natural language. In such cases, users describe their *intention* with natural language, and COA gathers potential *offers* and runs an auction to choose the best one. This means a human user can interact with the capability ecosystem without the constraints of specific applications—no matter where users are and what they're using, they're able to consume all capabilities in the ecosystem without switching contexts. Multitasking becomes a thing of the past because everything can happen in every context. Instead of switching between tasks or contexts, users are in a seamless, continuous flow.

When you design a system, don't think of services; think of capabilities. It might seem to be a subtle change, but you'll thank yourself later that you've made the switch.

You Can Cloudify Your Monolith

Jake Echanove

Senior VP for Solutions Architecture at Lemongrass Consulting

Application rationalization exercises often determine that monolithic workloads are better left on premises, insinuating that cloud benefits can't be realized. But monoliths don't have to be migrated to cloud native or microservices architectures to take advantage of cloud capabilities. Many methods can be employed to help legacy applications, such as SAP and Oracle apps, realize the agility, scalability, and metered billing advantages of the cloud.

First, it is important to have a deep understanding of the application architecture to ensure that the future landscape is flexible enough to be scalable. For instance, many applications employ architectures consisting of web servers, application servers, and databases. Sometimes these tiers are combined in single-instance deployments, which is a disadvantage in the cloud. If the tiers are combined on a non-86 platform, they should be separated when migrating to an x86-based cloud platform. This will help ensure that the web, app, and database tiers are loosely coupled and can grow and shrink without affecting the other tiers.

Second, it is key to be able identify and understand workload tendencies. Let's take an enterprise resource planning (ERP) financial system as an example. The month-end close is a busy time for the system, because many users are running reports, running close scenarios, and performing other activities occurring only at the month's end. Other times of the month are less busy, thus requiring less resources. In the cloud, administrators can bring up extra application servers at month's end and shut them down for the rest of the month to save on costs or reallocate resources for other purposes. Having knowledge of workload characteristics is key to help admins understand when to scale to meet requirements and when to shut down systems to save on costs.

Third, it is imperative to know that automation isn't just for cloud native applications. Scaling monolithic applications without user intervention is possible if the cloud admin understands the inner workings of the application. It is common knowledge that autoscaling is often used with cloud native technologies. For example, cloud native apps may be monitored for metrics such as high CPU utilization and then can trigger an event to deploy a new container to spread the workload. Legacy applications often require a different approach, because they don't function with containers or leverage microservices. The work processes within the application would need to be monitored. This is not merely monitoring an OS process, but interfacing at the application layer to determine whether the application is taxed. If so, the next step would be to trigger an event to spawn additional application servers. It is also possible to recognize a workload decrease to then safely shut down application servers without losing transactions.

Last, advanced methods can create DevOps-like deployment models, use AIOps methodologies for day 2 support, and extend the legacy core functionality using a microservices architecture. Many customers have deployed these methods into their production landscapes to make their legacy apps more cloud native–like, but deploying some of these operating models requires a shift in mindset and a deep understanding of the applications being moved to the cloud. The possibilities are extensive for those cloud admins who also possess application expertise with legacy workloads or that work closely with application owners.

Integrating Microservices in Cloud Native Architecture

Kasun Indrasiri

Product Manager and Senior Director at WSO2

When we construct cloud-based applications, we embrace cloud native architecture in the design to meet agility, scalability, and resiliency requirements. A cloud native application is designed as a collection of microservices that are built around business capabilities.

These microservices interact with each other and with external applications through interprocess communication techniques. These interactions can range from invoking other microservices to creating composite microservices by combining multiple microservices and other systems, building an event consumer or producer service leveraging an event/message broker, creating a microservice facade for a legacy monolithic system, and so on. The process of building the interactions between these microservices is known as *microservices integration*.

The integration of services, data, and systems has long been a challenging yet essential requirement in the context of enterprise software application development. In the past, we used to integrate all of these disparate applications using a point-to-point style, which was later replaced by a centralized integration middleware layer known as an enterprise service bus (ESB) with service-oriented architecture (SOA). Here the ESB acts as the middleware layer that provides all the required abstractions and utilities to integrate other systems and services. But in the cloud native era, we no longer use a central, monolithic shared layer containing all our integration logic. Rather, we build microservice integrations as part of the microservice's business logic itself.

For example, suppose you are designing an online retail application using a microservices architecture and you have to develop a checkout service that needs to integrate with other services: inventory, shipping, and a monolithic enterprise resource planning application. In the ESB era, you would have

developed the checkout service as part of the ESB by plumbing in all the required services and systems. But in the context of microservices, you don't have an ESB, so you build all the business and integration logic as part of the checkout service's business logic.

If we take a closer look at microservice integration logic, one portion of that logic is directly related to the business logic of the service while the other portion is pretty much about interprocess communication. For instance, in our example, the composition logic where we invoke and compose the responses of all the downstream services and systems is part of the business logic of the checkout service, and the network communication between the services and systems (using techniques such as circuit breakers, retries, wire-level security, and publishing data to observability tools) is agnostic of the business logic of the service. Having to deal with this much complexity as part of microservice development persuades us to separate the commodity features that we built as part of the network communication from the service's business logic.

This is where a service mesh comes into the picture. A *service mesh* is an interservice communication layer where you can offload all the network communication logic of the microservices you develop. In the service mesh paradigm, you have a colocated runtime, known as a *sidecar*, along with each service you develop. All the network communication–related features, such as circuit breakers and secured communication, are facilitated by the sidecar component of the service mesh and can be centrally controlled via the service mesh control plane.

With the growing adoption of Kubernetes, service mesh implementations (such as Istio and Linkerd) are increasingly becoming key components of cloud native applications. However, the idea that a service mesh is an alternative to the ESB in a microservices context is a common misconception. As mentioned previously, it caters to a specific aspect of microservice integration: network communication. The business logic related to invoking multiple services and building composition still needs to be part of the service's business logic. Also, we need to keep in mind that most of the existing implementations of the service mesh are designed only for synchronous request/response communication. The concepts used in the service mesh and sidecar architecture have been further developed to build solutions such as Dapr, where you can use a sidecar that can be used for messaging, state management, resilient communication, and so on.

To cater to the requirements of microservices integration and help you avoid building all these complex integrations from scratch, various cloud native

integration frameworks are available, such as Apache Camel K, Micronaut, and WSO2 Micro Integrator. When you develop a cloud native application, based on the nature of the microservice that you're developing, you can use such an integration framework to build your microservice while leveraging the service mesh for all the network communication–related requirements.

Containers Aren't Magic

Katie McLaughlin

Developer Advocate at Google Cloud

Containers, and the Open Container Initiative (OCI) image format specifications, aren't magic cure-alls. Popularized by Docker in the mid-2010s, the concept of having a definition create an isolated space for software to live in isn't unique and isn't a panacea.

Isolation standards have existed for years: virtual machines (VMs) are an isolation mechanism. Your VM may not touch your neighbor's VM, unless you specifically allow it to (typically, through network firewalls). However, that doesn't mean you can't have vulnerable software and malicious programs on your system.

Containers can be seen as just an iteration on VMs, but in a smaller form factor. VMs allowed more isolation environments on bare-metal servers. Containers serve the same purpose, and are vulnerable to the same issues as VMs. Indeed, Docker itself has been shown to be reproducible in a mere 100 lines of bash. The mechanisms by which we achieve isolation are not unique or new, and the advantages they give us don't outweigh the considerations we need to keep in mind.

Downloading a random executable from the internet without knowing what it does and running it on your local machine is something that should cause a tickle in the back of any programmer's head. So why would you include a `FROM` in your Dockerfile without knowing the origin? If you can't see the source of the image you're downloading from Docker Hub, anything could be in there.

Just as containers can contain anything to start with, the packages that they are intended to contain won't always be benign. In any given month multiple vulnerability websites may be launched, using cute logos and punny names for at least a vague semblance of a marketing strategy. All this effort is not just to draw attention to the researchers who find the issues (though it helps), but to make sure everyone who runs the affected systems and needs to apply fixes knows about them and can adjust their systems accordingly.

But while vendors can update the operating systems of the hardware that's hosting container platforms to apply security patches, the contents within the containers can be the issue. Throwing a bunch of legacy code in a container to "keep it safe" won't help when the code itself contains something that no container isolation environment can prevent escaping.

Using container scanning services to periodically check the contents of your images for the latest known issues is just one way to ensure that you know of any problems as soon as possible—implementing security standards when the images are created is a better defense.

For containers that don't require complex system calls, running these in a strict container sandbox—a system that itself can't call destructive commands—may be the best way to go. That way, even if your Eldritch horror escapes its confinement, the damage it can do is minimal. You can't call system commands that don't exist.

You can create secure containers that you can't break out of, but this requires effort, security awareness, and constant vigilance.

Your CIO Wants to Replatform Only Once

Kendall Miller

President of Fairwinds

When you work for a small consultancy that helps companies modernize their infrastructure, you get the rare opportunity to touch many kinds of infrastructure. When you arrive at a client site, you can roll up your sleeves, get knee deep in unbelievably rotten spaghetti infrastructure code, and decide to put in place something you know is considered best practice but that you've never gotten to play with before. And then, uniquely, on the next engagement, you can try something else that's new: you've heard about an *even shinier* way to do infrastructure, so you want to go chase that squirrel. Learning is fascinating, and solutions will continue to evolve.

If you work for a product company, however, and if your infrastructure has been replatformed in the last five years, your CIO (or CTO, or VP of engineering, or…uh…marketing director—let's be honest, it happens) is going to have zero patience for a new platform for the sake of a new platform. You might be bored out of your mind, or your pager might be going off every night for totally fixable reasons. But that doesn't mean your leadership team has the capacity to stomach a change in tooling or believes that "pouring a gallon of Kubernetes on things" is going to make all of your problems go away.

So, if you're an engineer stuck with an age-old infrastructure, you need a strategy for picking and then pitching a replatforming.

When you pitch your proposal to the executives in charge, include the reasons why this particular change will increase velocity, why it will help you ship faster, and—for goodness' sake—why it will be the last replatforming the company needs to do for another 5 to 10 years. The pitch needs to cover a wide range of factors proving that your suggestion offers that Goldilocks combination of ease and flexibility. You need to almost sing and dance about how your changes will reduce the need for new headcount and enable scalability, security, efficiency, and future proofing.

Then, pick something with tremendous community backing. Today that's Kubernetes; in a few years it may be something different, or something built on top of Kubernetes. Kelsey Hightower (a principal engineer at Google) once said, "I'm convinced the majority of people managing infrastructure just want a PaaS. The only requirement: it has to be built by them." Kubernetes today is the ultimate PaaS builder, but it also enables something as close to cloud agnosticism as is possible right now. Your CIO will love the words *cloud agnosticism*.

As a systems engineer at a product company, your desire to learn can sometimes feel like it's in direct conflict with your CIO's desire for stability. Understanding that the company wants to replatform only once (and all the incentives this directly impacts) is the only hope you have for a successful pitch to make that replatforming happen during your tenure.

Everyone deals with infrastructure spaghetti code (if they're lucky enough to find infrastructure as code at all), whether it's the engineer who wrote that code or the one who builds the platform everything runs on. Get rid of it by convincing your CIO that this is the time, and you are the person, for that once-ever replatforming.

Practice Visualizing Distributed Systems

Kim Schlesinger

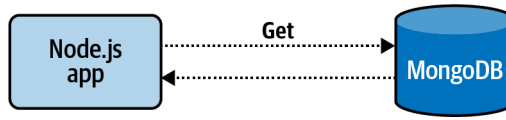
Site Reliability Engineer at Fairwinds

Before cloud computing, ops engineers were more likely to have seen, held, and physically maintained their servers. The primary reason for the industry-wide shift from on-premises datacenters to cloud service providers is that cloud providers carry the burden of maintaining the physical computers and hardware they rent, which allows cloud engineers to focus on designing cloud infrastructure, continuous integration and continuous delivery (CI/CD) pipelines, and applications. It also means that servers are far away and invisible.

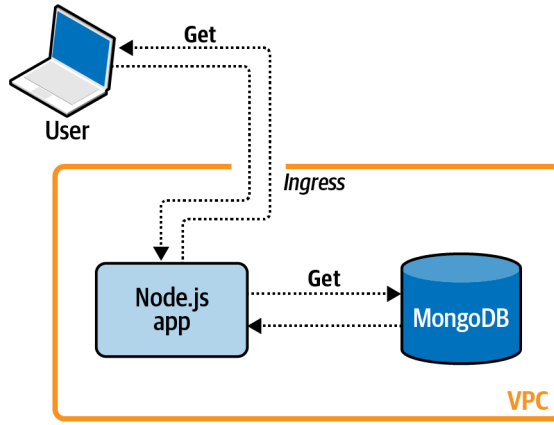
Highly skilled cloud engineers are able to imagine parts of the systems they build and maintain, and they can visualize how a request flows from one component to another. This isn't a skill most of us are born with, but with determination and practice, you can begin imagining and understanding your invisible systems, and this will help you be a better engineer.

While there are several ways to begin visualizing your cloud infrastructure, no matter the path you take, it is important that you construct these visualizations yourself, not just look at diagrams or graphs created by someone else. The act of wrestling part of your system into a diagram or model will be the fastest path to understanding, even if your model isn't perfect.

Start with a part of your distributed system that has two or three components. For example, imagine you have a Node.js application that stores and retrieves data from a MongoDB database, and both components are deployed as containers on two separate instances on a public cloud like AWS. A quick way to start visualizing is by drawing these parts as a block diagram and showing the HTTP requests as arrows.



As you draw this diagram, you will likely ask yourself, "How is the initial request from the user getting from the internet to my application, which is inside a virtual private cloud (VPC)?" Then you add the virtual private cloud and ingress.



You could add regions and availability zones, Secure Sockets Layer (SSL) termination, the flow of authentication and authorization, replicas, load balancers, and more until your diagram is the size of a small city, but that's not the point, and you have to stop eventually. The goal of this exercise is to make sense of one part of your system, and through that understanding you are freeing up your cognitive energy to improve that part or to begin understanding something else.

Block diagrams are an easy way to get started, but they are limited by their two dimensions. Other tools include data visualizations like those in the D3.js library, web sequence diagrams that show how requests play out over time, and physical 3D models like the solar system you built in the fourth grade. The 3D model takes a lot of time and effort to build, but it's fun as hell, and you can start to feel out the size of components, how "far away" they are from each other, and the states they share (or don't), like memory and the network.

Being able to imagine your distributed systems will help you suss out cause-and-effect relationships that will make your debugging (and response to incidents!) faster and more accurate. Once you do two or three visualization exercises, you will start identifying cloud infrastructure patterns that you can apply in a more senior role like cloud architect. If you practice visualizing with your team, you'll have valuable debates about the best way to model your system, which will increase your team's collective understanding. Finally, if you practice visualizing your distributed systems, your monitoring graphs and observability tools will be a rich layer of data in addition to your strong understanding of your cloud infrastructure and your applications.

Cloud engineers have superpowers. We can change one line of configuration code and turn off a computer on another continent, or run a command that will quadruple the number of nodes, unlocking access to an application for users from all over the world. Being able to manipulate machines that are unseen is a wizard's trick, but it also makes our systems opaque and hard to understand. It's worth your time to begin understanding your cloud infrastructure by practicing how to visualize distributed systems.

Know Where to Scale

Lisa Huynh

Lead Software Engineer at Storyblocks

Most of the time, if all goes well, an application will hit a point where it needs to grow. Outside of "the application is timing out," however, determining an acceptable level of performance can be subjective. Someone whose customers are all in Canada may not care about response times in Japan.

Whatever your metrics, let's say you're there. Typically, we can upgrade our systems by scaling up or out, also called vertical and horizontal scaling, respectively. With *vertical scaling*, we upgrade a resource in our existing infrastructure. With *horizontal scaling*, we add more instances. But which one should you be doing?

Vertical Scaling

You're hitting the CPU limit, so you upgrade your instance from one with 8 CPUs to 16. Or maybe you're running out of storage space, so you go from 100 GiB to 500 GiB. With this easiest and simplest way to scale, you're running the same application but on more powerful resources.

Most relational databases use vertical scaling so that they can guarantee data validity (atomicity, consistency, isolation, and durability, or ACID properties) and support transactions. So, for applications requiring a consistent view of the data, such as in banking, you'd typically stick to vertical scaling.

Unfortunately, this type of scaling often involves downtime. If there's nowhere else to divert traffic, customers have to wait while the instance is upgraded. Hardware also gets expensive and has its limits.

Horizontal Scaling

On the other hand, if your application is stateless or works with eventual consistency, you can use horizontal scaling. Just increase the number of machines that run the application and distribute the work among them. This

is great for handling dynamic loads, such as normal fluctuations throughout the day, and avoiding the "hug of death" from a surge of traffic.

If your application relies on state, you may need to change it in order to use horizontal scaling. NoSQL databases can scale out because they make trade-offs of weaker consistency; during updates, invalid data could be returned. Also, you will need some way to distribute traffic across your instances. The application could handle this itself, or a dedicated load balancer resource could handle routing the traffic to instances.

Smart routing should also bring reliability and allow changes without downtime, as it should avoid "bad" or unavailable instances. Numerous policies can be applied to more smartly shape your traffic. For instance, if you're trying to serve a global audience, you may end up being constrained by the time it takes for requests to travel from an end user to your server. In that case, you may consider adding instances in multiple regions, and you'll need a balancer that will route to the closest available server.

If you are attempting to improve the response time of *static assets* (such as HTML pages), consider specialized services called *content delivery networks* (CDNs). A CDN handles distributing your assets across its global network of servers and routing each end user to the most optimal server. This can be a lot simpler than building out that network yourself.

Conclusion

In the end, which strategy you use to scale will depend on your system's requirements and bottlenecks. As a rule of thumb, vertical scaling is simpler to manage and great for applications requiring atomicity and consistency—but upgrading can be expensive and require downtime. Horizontal scaling is elastic and brings reliability, yet is more complicated to manage.

Serverless Bad Practices

Manasés Jesús Galindo Bello

Software Architect at Cumulocity IoT by Software AG

Amazon's launch of AWS Lambda, launched in 2014 made it the first cloud provider with an abstract serverless computing offering. Serverless is the newest approach to cloud computing, enabling developers to run event-driven functions in the cloud without having to administer the underlying infrastructure or set up the runtime environment. The cloud provider manages deployment, scaling, and billing of the deployed functions.

Serverless has become a buzzword that attracts developers and cloud engineers. The most relevant implementation of serverless computing is the function as a service (FaaS). When using a FaaS, developers only have to deploy the code of the functions and choose which events will trigger them. Although it sounds like a straightforward process, certain aspects have to be considered when developing production-ready applications, thus avoiding the implementation of a complex system.

Deploying a Lot of Functions

FaaS follows the pay-as-you-go approach; deployed functions are billed only when they are run. As there are no costs for inactive serverless functions, deploying as many functions as you want might be tempting. Nevertheless, this may not be the best approach, as it increases the size of the system and its complexity—not to mention that maintenance becomes more difficult. Instead, analyze whether there is a need for a new function; you may be able to modify an existing function to match the change in the requirements, but make sure it does not break its current functionality.

Calling a Function Synchronously

Calling a function synchronously increases debugging complexity, and the isolation of the implemented feature is lost. The cost also increases if the two functions are being run at the same time (synchronously). If the second function is not used anywhere else, combine the two functions into one instead.

Calling a Function Asynchronously

It is well known that asynchronous calls increase the complexity of a system. Costs will increase, as a response channel and a serverless message queue will be required to notify the caller when an operation has been completed. Nevertheless, calling a function asynchronously can be a feasible approach for one-time operations; e.g., to run a long process such as a backup in the background.

Employing Many Libraries

There is a limit to the image size, and employing many libraries increases the size of the application. The warm-up time will increase if the image size limit is reached. To avoid this, employ only the necessary libraries. If library X offers functionality A, and library Y offers functionality B, spend time investigating whether a library Z exists that offers A and B.

Using Many Technologies

Using too many frameworks, libraries, and programming languages can be costly in the long term, as it requires people with skills in all of them. This approach also increases the complexity of the system, its maintenance, and its documentation. Try limiting the use of different technologies, especially those that do not have a broad developer community and a well-documented API.

Not Documenting Functions

Failing to document functions is the bad practice of all times. Some people say that good code is like a good joke—it needs no explanation. However, this is not always the case. Functions can have a certain level of complexity, and the people maintaining them may not always be there. Hence, documenting a function is always a good idea. Future developers working on the system and maintaining the functions will be happy you did it.

Getting Started with AWS Lambda

Marko Sluga

Cloud Consultant and Instructor

AWS Lambda is a serverless processing service in AWS. When programming with Lambda, a logical layout of your application is literally all you need. You simply need to make sure each component in the layout maps directly to a function that can independently perform exactly one task. For each component, code is then developed and deployed as a separate Lambda function.

AWS Lambda natively supports running any Java, Go, PowerShell, Node.js, C#, Python, or Ruby code package that can contain all kinds of extensions, prerequisites, and libraries—even custom ones. On top of that, Lambda even supports running custom interpreters within a Lambda execution environment through the use of layers.

The code is packaged into a standard ZIP or WAR format and added to the Lambda function definition, which in turn stores it in an AWS-managed S3 bucket. You can also provide an S3 key directly to Lambda, or you can author your functions in the browser in the Lambda section of the AWS Management Console. Each Lambda function is configured with a memory capacity. The scaling of capacity goes from 128 MB to 3,008 MB, in 64 MB increments.

The Lambda section of the Management Console allows you to manage your functions in an easy-to-use interface with a simple and efficient editor for writing or pasting in code. The following example shows how to create a simple Node.js Lambda function that prints out a JSON-formatted response after you input names as key/value pairs.

Building an Event Handler and Testing the Lambda Function

Start by opening the AWS Management Console, going to the AWS Lambda section, and clicking Function and then "Create function."

Next, replace the default code with the code shown here. This code defines the variables for the key/value pairs you will be entering in your test procedure and returns them as JSON-formatted values:

```
exports.handler = async (event) => {
var myname1 = event.name1;
var myname2 = event.name2;
var myname3 = event.name3;
var item = {};
item [myname1] = event.name1;
item [myname2] = event.name2;
item [myname3] = event.name3;
const response = {
body: [ JSON.stringify('Names:'), JSON.stringify(myname1), JSON.
stringify(myname2), JSON.stringify(myname3)],
};
return response;
};
```

When you are done creating the function, click the Save button at the top right.

Next, you need to configure a test event for entering your key/value pairs. You can use the following code to create your test data:

```
{
"name1": "jenny",
"name2": "simon",
"name3": "lee"
}
```

Once you've entered that, scroll down and click Save at the bottom of the Configure Test Event dialog box. Next, run the test, which invokes the function with your test data. The response should be a JSON-formatted column with the value `Names`, and then a list of the names that you entered as test data.

In the execution result, you also have information about the number of resources the function consumed, the request ID, and the billed time. At the bottom, you can click the "Click here" link to go to the logs emitted by Lambda into Amazon CloudWatch.

In CloudWatch, you can click the log stream and see the events. By expanding each event, you get more detail about the request and duration of the execution of the Lambda function. Lambda also outputs any logs created by your code into this stream because the execution environment is stateless by default.

In this example, you've seen how easy it is to create, deploy, and monitor an AWS Lambda function—and that serverless truly is the future of cloud computing. Enjoy coding!

It's OK if You're Not Running Kubernetes

Mattias Geniar

Cofounder of Oh Dear

I love technology.[1] We're in an industry that is fast-paced, ever improving, and loves to be cutting-edge and bold. It's this very drive that gives us exciting new tech like HTTP/3, Kubernetes, Golang, and so many other interesting projects.

But I also love stability, predictability, and reliability. And that's why I'm here to say that it's OK if you're not running the very latest flavor-du-jour insert-new-project-here.

The Media Tells Us Only Half the Truth

If you were to read or listen to only the media headlines, you might believe everyone is running their applications on top of an autoscaling, load-balanced, geo-distributed Kubernetes cluster backed by only a handful of developers who set the whole thing up overnight. It was an instant success!

Well, no. That's not how it works. The reality is, most Linux or open source applications today still run on a traditional Debian, Ubuntu, or CentOS server—as a VM or a physical server.

I've managed thousands of servers over my lifetime and have watched technologies come and go. Today, Kubernetes is very hot. A few years ago, it was OpenStack. Go back some more, and you'll find KVM and Xen, paravirtualization, and plenty more.

I'm not saying all these technologies will vanish—far from it. Each project or tool has merit; they all solve particular problems. If your organization can benefit from something that can be fixed that way, great!

1 A version of this article was originally published at SYSADVENT.

There's Still Much to Improve on the Old and Boring Side of Technology

My background is mostly in PHP. We started out using the Common Gateway Interface (CGI) and FastCGI to run our PHP applications and have since moved from mod_php to php-fpm. For many system administrators, that's where it ended.

But there's so much room for improvements here. The same applies to Python, Node.js, or Ruby. We can further optimize our old and boring setups (you know, the ones being used by 90% of the web) and make them even safer, more performant, and more robust.

Were you able to check every configuration and parameter? What does that obscure setting do, exactly? What happens if you start sending malicious traffic to your box? Can you improve the performance of the OS scheduler? Are you monitoring everything you should be? That Linux server that runs your applications isn't finished. It requires maintenance, monitoring, upgrades, patches, interventions, backups, security fixes, troubleshooting…

Please don't let the media convince you that you should be running Kubernetes just because it's hot. You have servers running that you know still have room for improvements. They can be faster. They can be safer.

Get satisfaction in knowing that you're making a difference for your business and its developers because your servers are running as best they can. What you do matters, even if it looks like the industry has all gone for The Next Big Thing.

But Don't Sit Still

Don't take this as an excuse to stop looking for new projects or tools, though. Have you taken the time yet to look at Kubernetes? Do you think your business would benefit from such a system? Can everyone understand how it works? Its pitfalls?

Ask yourself the hard questions first. There's a reason organizations adopt new technologies. It's because they solve problems. You might have the same problems!

Every day new projects and tools come out. I know because I write a weekly newsletter about it. Make sure you stay up-to-date. Follow the news. If something looks interesting, try it! But don't be afraid to stick to the old and boring server setups if that's what your business requires.

Know Thy Topology

Nikhil Nanivadekar

Director at BNY Mellon

In today's era of cloud computing, it is imperative to understand the structure of a system. A holistic view of the system topology is important to understand how a system works and to figure out the multiple moving components. A few main aspects to consider are modularity, deployment strategy, and datacenter affinity.

Modularity

When creating a modular system, the simplest rule to follow is to ensure separation of concerns between functionality. A particular microservice should be responsible for carrying out a single function and its related processing. This helps microservices have a small footprint. Microservice instances should be stateless, replaceable, and scalable. If a microservice instance is replaced by another instance of the same microservice, the output should be the same. If a microservice instance is scaled by adding more instances, the system should still function properly.

Deployment Strategy

Deployment means releasing a new version of an application to production. Deployment strategies need to be considered while upgrading an application, as they directly backward compatibility. Multiple deployment strategies are used today, including:

Re-create
: The old version is shut down; the new version is rolled out.

Rolling update (incremental)
: The new version is incrementally rolled out to replace the old version.

Blue/green
: The new version is fully released while the old version is working, and traffic is directed from the old version to the new version.

Canary
: The new version is released to a small group of users before releasing it broadly.

A/B testing
: In this extension of a canary deployment, the new version is released to a small group of users, and depending on the adoption of new features, the features are rolled out broadly.

Shadow (prod-parallel)
: The new version is released, and both the old and new versions serve the same requests.

In choosing a strategy, the key consideration is how many versions of the application need to be functional at the same time. If multiple versions of the same application need to run at the same time, maintaining backward compatibility is necessary.

Datacenter Affinity

In a multi-datacenter architecture, services run in multiple datacenters. This approach provides redundancy, disaster recovery, and scalability.

Depending on your needs, you may deploy every service or a only subset of the services. In an *all-active* model, all the deployed services across all datacenters serve the requests. In an *active–passive* model, one set of datacenters is deemed *active*, and a second set is deemed *passive*. All traffic is directed to services in the active datacenters. If a disaster recovery activity occurs and the datacenters need to be switched, in an active-passive scenario, all the requests from the active datacenters are directed to the passive datacenters.

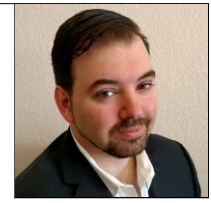
While a multi-datacenter architecture provides numerous benefits from a resiliency perspective, it can cause latency issues. Each time service-to-service calls occur, data needs to be transferred from one location to another. If the services are colocated, the data transmission time is minimal. If not, latency can be significant. Consider four services across two datacenters: datacenter 1 has services A and C, and datacenter 2 has services B and D. Assume there is a 5-millisecond transmission time between datacenters. So, if A calls B, B calls C, and C calls D, then the latency becomes 15 ms. If the same process is followed serially for 1,000 calls, the latency adds up to 15 seconds, thereby causing a slow response time.

In a distributed world, multiple moving components exist. The topology of the system becomes an important piece of the puzzle to control them. As long as you have a solid understanding of the topology, these multiple moving components can prove to be a boon when delivering software solutions.

System Fundamentals Will Still Bite You

Noah Abrahams

Director of Enablement



Imagine you've just joined a new company, ready to embark on your first day of cloud engineering. You've been reading all about DevOps principles, and you're excited to get started in a world dominated by APIs and GitOps. This level of technology comes with the promise of focusing on only what's directly relevant to your role, while everything else is just "taken care of" for you. You go to push your first deployment and…the job fails? It hits…an issue with an upstream repository? How is that even a thing? Doesn't all this stuff "just work"?

The most common mistake that I've seen among those getting ramped up with cloud computing is to assume that because these are commercially available solutions, all the possible underlying issues are somehow magically taken care of. Being available for your use absolutely does *not* mean that the systems you're given are free from bugs or idiosyncrasies. I've spent the past few years working at a layer of container orchestration that is multiple levels of abstraction away from what would otherwise be traditional "system management," and it can really make you feel like you're working in another realm entirely—so it's important to not lose sight of what your stack is built from.

Kubernetes pods aren't terminating properly? Maybe that's not an API or etcd issue, but instead you've just stumbled across a problem with systemd not handling the containerd shims correctly. Congratulations on finding an obscure bug. Containers are crashing on bootup, but they ran fine on your development system? Oh, look, you just hit a file descriptor limit in the AMI's default kernel settings. Now you need to change that as part of your boot process. Apparently running out of IPs, but you can't figure out why or how? Maybe it's an issue with the hypervisor underneath the instance type you chose and the number of devices it can natively handle. A cascading node-by-node failure across your whole platform? Maybe you should check

your certificate expiration dates. Network connections aren't working as expected? There's an 85% chance it's actually a DNS problem.

This isn't to say that the upper layers don't also have their fair share of problems, but if you're an operations engineer, think about this: how many times have you been annoyed by a development team blaming your shiny new platform because a database connection string didn't work? Since your system was the new piece, the problem "*must* be on your end." You knew it was a problem with their SQL connection, but you had to convince them that the problem was there. The only way to do that was to examine the problem at a fundamental layer. Why would you treat your own systems any differently?

The point is, building your application or platform on top of the abstractions that a cloud provider gives you does not make the underlying layers stop existing. In many cases, it makes them even more important. An issue at, for example, the OS image level can cascade to affect every layer above it, and manifest across your whole distributed system. In many cases, the failure modes can also be less obvious than we're used to. Therefore, having good visibility into your systems (through, for example, logging with minimal noise, using appropriate metrics, and having observability at all) will help separate the issues at the layer you're focused on from the fundamental system issues that can still bite you.

So, long story short: don't forget what's under the hood; you will probably still have to deal with the things you expect to be abstracted away; and when in doubt, it's probably DNS.

Cloud Processing Is Not About Speed

Rustem Feyzkhanov

Business Information Security Officer

Data and machine learning processing pipelines used to be about speed. Now we live in a world of public cloud technologies, where any company can procure additional resources in a matter of seconds. This fact changes our perspective on the way processing pipelines should be constructed.

In practice, we pay equally for providing 10 servers for 1 minute or 1 server for 10 minutes. Therefore, the focus shifts from optimizing execution time to optimizing scalability and parallelization.

Let's imagine a perfect data processing pipeline: 1,000 jobs get processed in parallel on 1,000 nodes, and then the results are gathered together. This would mean that at any scale, the speed of processing doesn't depend on the number of jobs and is always equal to the processing time for one job.

This doesn't sound so impossible. Serverless infrastructure, which is becoming more and more popular, provides a way to launch thousands of processing nodes in parallel. In addition, more and more vendors now have pure container-as-a-service offerings, meaning that once you define, for example, a Docker image, it will be executed in parallel, and you will pay for only processing time. Not only that, but when serverless infrastructure and container as a service are coupled with native message buses or orchestrators they are able to handle large numbers of incoming messages by independently mapping them to these scalable computer services. These services enable a lot of opportunities, and by utilizing them we can minimize idle time and scale infrastructure to match load perfectly.

But once we achieve perfect horizontal scalability, should we focus on execution time? Yes, but for a different reason. In a world of perfect horizontal scalability, execution time doesn't influence the speed of processing the batch much, but it significantly influences the cost. Optimizing speed twice means optimizing cost twice, and that is the new motivation for optimizing development.

Furthermore, designing an absolutely scalable data pipeline without taking into account optimizing algorithms at all can lead to an extremely high cost of the pipeline. That is one of the disadvantages of a system that doesn't have an economy of scale.

One additional opportunity is about separating apps into modular parts and executing each part on a separate scalable service. This approach provides a way to find the best fit for your application and minimize idle CPU (or GPU) and RAM. Once we find this kind of fit, we can not only minimize the cost of processing, but also make sure that we process each part fast enough (for example, we might preprocess data not on the GPU VM, but in parallel on multiple CPU VMs). And finally, we can start using CPU VMs from one vendor, GPU VMs from another vendor, and serverless computing resources from a third one to find the best balance between speed, cost, and scalability.

The emerging opportunity is to design data pipelines to optimize unit costs and enable scalability from initial phases for transparent communication between data engineers and other stakeholders, such as project managers and data scientists.

How Serverless Simplifies the Developer Experience

Wietse Venema

Senior Google Cloud Engineer at Binx.io

If you use serverless components to build your application, your application is serverless—but what does *serverless* mean?[1] It's an abstract and overloaded term that means different things to different people.

When trying to understand serverless, you shouldn't focus too much on the "no servers" part—it's more than that. In general, this is what I think people mean when they call something *serverless*, and why they are excited about it:

- It simplifies the developer experience by eliminating the need to manage infrastructure.
- It's scalable out of the box.
- Its cost model can result in significant savings, because you pay exactly for what you use, not for capacity you reserve up front. If you use nothing, you pay nothing.

Here, I'll focus on the first characteristic. What does that simple developer experience mean?

Eliminating infrastructure management means you can focus on writing your code and have someone else worry about deploying, running, and scaling your application. The platform will take care of all the important and seemingly simple details that are surprisingly hard to get right. Examples include autoscaling, fault tolerance, logging, monitoring, upgrades, deployment, and failover.

One thing you specifically *don't* have to do in the serverless context is server management. Servers still exist in a serverless platform, but you don't have to

1 Excerpt from *Building Serverless Applications with Google Cloud Run* by Wietse Venema (O'Reilly, 2020).

worry about them anymore. The platform offers an abstraction layer. This is the primary reason we call it *serverless*.

When you are running a small system, server management might not seem like a big deal, but readers who manage more than 10 servers know that this can be a significant responsibility that takes a lot of work to get right. Here is an incomplete list of tasks you no longer need to do when you run your application logic on a serverless platform:

- Provisioning and configuring servers (or setting up automation)
- Applying security patches to your servers
- Configuring networking and firewalls
- Setting up Secure Sockets Layer/Transport Layer Security (SSL/TLS) certificates, updating them yearly, and configuring a web server
- Automating application deployment on a cluster of the servers
- Setting up logging and metrics monitoring to provide insights into system performance

And that is just about servers. Most businesses have higher and higher expectations for system availability. More than 30 minutes of downtime per month is generally not acceptable. To reach these levels of availability, you will need to automate your way out of every failure mode; there is not enough time for manual troubleshooting. As you can imagine, this is a lot of work and leads to more complexity in your infrastructure. If you build software in an enterprise environment, you'll have an easier time getting approvals from security and operations teams, because a lot of their responsibilities shift to the vendor.

Availability is also related to software deployments, now that it is more common to deploy new software versions on a daily basis instead of monthly. When you deploy your application, you don't want to experience downtime, even when the deployment fails.

Serverless technology helps you focus on solving your business problems and building a great product, while someone else worries about the fundamentals of running your app. This sounds convenient, but you shouldn't take this as *all your responsibilities disappear*. Most important, you still need to write and patch your code, and make sure it is secure and correct. You still need to manage some configuration, like setting resource requirements, adding scaling boundaries, and configuring access policies.

Serverless is great if you value a simple and fast developer experience, and if you don't want to build and maintain traditional server infrastructure. The servers are still there—you just can't manage them anymore.

PART III

Migration

People Will Expect Things—Help Them Expect Right

Dave Stanke

Developer Advocate at Google

News travels fast. When your stakeholders—your nontechnical colleagues, your customers, and everyone in between—learn that you're migrating to the cloud, they may react in a variety of ways:

> "You'll regret it. I had a friend who did cloud, and now he has no control over his expenses."
> "Sweet! All the tech billionaires run their stuff on the cloud. We're going to be rich!"
> "OK, but it'll be *your* head when we get hacked."

…and perhaps the most likely reaction of all:

> "Er, where's the cloud?"

You may be tempted to avoid the conversation entirely. You could decide that your migration is need-to-know, and that anyone who won't be hands-on for the transition doesn't need to know. That's a mistake. Why? Because all your stakeholders will be affected. Innumerable tiny and not-so-tiny differences exist between running your own datacenter and using a cloud provider. Every difference is likely to have an impact on user experience, leading to conversations like this:

> "Why isn't this the way it used to be?"
> "Because cloud."
> "Because *where*?"

Most importantly, remember that you're moving to the cloud for a reason. You're going to put in a lot of work to have a successful migration. (It *will* be successful. I believe in you.) Your stakeholders need to feel that it's worth it, rather than wishing you had spent that effort elsewhere. So…

Do engage nontechnical stakeholders in your migration. Inspire them to see the benefits of the cloud, and help them understand its transformative potential.

But *don't* oversell it. Maybe you're excited about rapid autoscaling. The cloud supports that! But it's not *magic*. Your application components won't instantly decouple themselves to acquire elasticity at every bottleneck. Or maybe you're planning to use a database as a service to reduce operational burden. Great! But it doesn't come with a free DBA on call to fix your busted schema. And while the cloud might enable automated provisioning for faster feature releases, someone has to put in the work to build those pipelines (and maybe write some tests while you're at it, hmm?).

On the other hand, *don't* undersell it. If you frame your migration as just a minor technical refactor, people will expect it to take a minor amount of effort. They'll also be resentful of any interruptions to their workflow. (Even good changes become bad news when people aren't expecting them.) You need to help them understand that this *is* a big deal. The cloud truly is a revolutionary technology, supporting entirely new ways of delivering software. To master it requires learning many new things, while *un*learning many old things. Teach stakeholders the possibilities, describe the journey, and dream big, together.

While you're at it, *do* share ongoing updates, and *do* make them relevant to your audience. Your cloud migration isn't likely to happen in an afternoon. It may take years. Design your migration to produce demonstrable stakeholder benefits, early and often. Then communicate those benefits as soon as they materialize. Your colleagues will be happy to receive status updates like "We turned down two server racks today, which will save an estimated $3,900 per month" or "Since we implemented blue/green deployments last month, we've caught two major bugs before they could have any customer impact."

Finally, *do* ask stakeholders for feedback and questions. The cloud is a mysterious place (well, "place"), and if you don't ask, you'll never know what assumptions—or misperceptions—they might have.

Learn what they expect, and help them expect right.

Failing a Cloud Migration

Lee Atchison

Cloud Consultant and author of Architecting for Scale

Moving to the cloud may appear easy, but you can make many common mistakes that can cause your migration to falter, struggle, or fail outright. Two of the most common mistakes that I see people make that hurt their ability to be successful in the cloud are not optimizing for the cloud and lacking an architectural strategy.

Mistake 1: Not Optimizing for the Cloud

Ever hear of a lift-and-shift migration? Moving an application to the cloud as is, without making any significant changes to the application, is a common novice migration strategy. This strategy is an easy way to move to the cloud, but more often than not, the benefits you expect to see from using the cloud will not materialize from a simple lift-and-shift migration.

For example, many people believe moving to the cloud will save money in terms of infrastructure costs. This is true for most organizations. However, organizations that perform a simple lift-and-shift often see their infrastructure costs increase after moving to the cloud.

Why is this so? In most cases, the increase is due to improper or incomplete planning before and during the cloud migration. Many of the cloud's cost benefits come from its ability to dynamically allocate and consume resources on demand, and then free the same resources when they are no longer needed. This powerful capability lets the cloud handle the scaling needs of your applications without requiring it to keep a significant quantity of spare resources in reserve to handle peak loads.

However, utilizing dynamic resources typically requires changes in your application architecture. Sometimes those changes are simple, and sometimes they are complex. But either way, if you perform a basic lift-and-shift migration and do not implement the necessary architectural changes, you end up utilizing the cloud with the same static processes that were in use on premises. This not only eliminates a critical financial benefit of using the cloud, but also can increase your overall costs. The cloud is not optimized

for replacing static infrastructures, so your costs may be unexpectedly higher than they were on premises.

Without properly optimizing for the cloud, you risk failing to meet your objectives for the cloud migration.

Mistake 2: Lack of Architectural Strategy

Even when migrating applications that are mostly ready for the cloud, significant technical planning is required. You still have to deal with migrating data, downtime management, and interservice latency during the migration. These issues require planning and management.

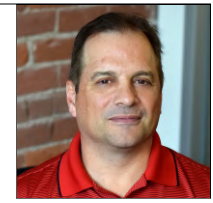
This is why I recommend that every organization performing a cloud migration create a *migration architect* role within the company. The person in this role should be the single point of decision making for all technical aspects of the migration and handle all the planning and rearchitecting needed to make the migration successful. This role can be a full-time job for a large migration, or part of a broader architecture role for a smaller migration—but having a single, clearly defined point of contact with responsibility for all technical decision making is critical.

Moving to the cloud isn't always painless, but it doesn't have to be painful. Avoiding these two common mistakes is an essential part of making your migration smooth and having it meet everyone's expectations.

Optimizing Processes for the Cloud: Patterns and Antipatterns

Mike Kavis

Managing Director of Technology/Cloud Practice, Deloitte Consulting



I've been consulting on cloud adoption since 2013.[1] Back then, convincing CEOs and boards that cloud computing was the way forward was a hard sell for IT leaders. But as public cloud adoption increased, companies moved to the cloud or built new workloads in the cloud much faster than they had traditionally deployed software. Two common antipatterns emerged.

Antipattern 1: The Wild West

Developers, business units, and product teams now had access to on-demand infrastructure, and they leveraged it to get their products out the door faster than ever. They had no guidelines or best practices, and development teams took on responsibilities they'd never had before. Rather than developing a systematic approach and implementing it across the organization, though, many companies simply left cloud decisions to individual parts of the organization: a lawless, Wild West approach.

Part of the problem was that each business unit or product team was "reinventing the wheel": each would research, buy, and implement its favorite third-party tools for logging, monitoring, and security. Each took a different approach to designing and securing the environment. More than a few also implemented their own continuous integration and continuous delivery (CI/CD) toolchains with very different processes, resulting in a patchwork of tools, vendors, and workflows throughout the same organization.

1 Excerpt from *Accelerating Cloud Adoption: Optimizing the Enterprise for Speed and Agility* (O'Reilly, 2021).

Companies were delivering value to their customers faster than ever before—but often exposing themselves to more security and governance risks than before, as well as delivering less-resilient products. Production environments became unpredictable and unmanageable because of a lack of rigor and governance.

Antipattern 2: Command and Control

The opposite of the freewheeling Wild West antipattern was a military-style, top-down, command-and-control approach. In these companies, units that were highly motivated to keep things in line—such as management, infrastructure, security, and governance, risk, and compliance (GRC) teams—put the brakes on public cloud access. They built heavily locked-down cloud services and processes that made developing software in the cloud cumbersome. These processes were often decades old, designed during the period when deployments occurred two or three times a year and all infrastructure consisted of physical machines owned by a separate team.

This approach destroys one of the key value propositions of the cloud: agility. I have seen companies that took six months to provision a virtual machine in the cloud—something that should take five minutes—because the command-and-control cops forced cloud developers to go through the same ticketing and approval processes required in the datacenter.

This approach created little value even as companies spent huge sums on strategy and policy work, building internal platforms that did not meet developers' needs. Worse yet, the approach created an insurgent "shadow IT": groups or teams began running their own mini-IT organizations to get things done, because their needs were not being met through official channels.

Avoiding Antipatterns

These antipatterns have raised awareness of the need to focus on cloud operations and to design a new cloud operating model.

When you move to the cloud for the first time, you are moving to a greenfield virtual datacenter. There are no processes in place. This is a one-time opportunity to design processes from the ground up, optimized for the cloud and its new ways of thinking. You'll never have a better chance to get this right. Don't simply bring your legacy processes and mindsets along for the ride. Your company's needs—all those security policies, compliance controls,

and operational requirements—are still valid; it's just how you satisfy them that needs to change.

Failing to acknowledge that legacy processes designed for another era aren't the best way to deliver software in the cloud will most likely result in low performance. This error in judgment will compound as more workloads move to the cloud, which can result in catastrophic consequences such as risk exposure, missed SLAs, and cost overruns. Transforming a culture to be more DevOps-centric starts with good process hygiene. Pick a process pain point and optimize it for the cloud. All the technology in the world can't fix bad processes.

Why the Lift-and-Shift Model Is Unlikely to Succeed

Mike Silverman

Head of Strategy at FS-ISAC

Someone tells you to move your workloads from on premises to a cloud service provider (CSP): "It will be easy; just lift and shift." Simple, right?

The short answer is no. A CSP and your on-premises environment are very different. Now you may be thinking, well, duh, you move to a CSP because there is some capability or feature that you can't achieve on premises—like elasticity, resilience, global reach, or economics. All of those are good reasons to consider a move to a CSP! (Each is discussed elsewhere in this book.) But when it comes to operating workloads in CSPs versus on premises, the way they're managed is quite different.

Consider logging as an example: the output from a CSP will likely differ from what you get from your on-prem hypervisor (or related systems). Can you commingle the logs? Unlikely. Perhaps you can map CSP events to your existing format and keep the same logging infrastructure. If you can, do you have different rules for on prem versus CSP (probably), and different automation or call routing? Does your application need to provide different event output if running in a CSP versus on prem? All things to consider, develop, test, and refine.

But that example assumes something that's probably untrue—that the CSP can access your logging systems! Those logs are typically behind firewalls (for good reason). Do you have connectivity from the CSP to behind your firewall? That can take a while to set up and configure.

As for the application itself, you need to ask whether it will run solely in the CSP or as a hybrid for some time. If hybrid, what key databases must you share across the CSP and on prem, what connectivity do you need between the two, and what is an acceptable latency? What about testing in the new architectures? How many more scenarios must you consider?

Separately, how do you move your application from on prem to the CSP? Where are your source code or packages/artifacts stored today? If on prem, can you move those to the CSP for deployment? It's not always easy to do that; you sometimes need different configurations or builds for CSPs.

Many considerations remain beyond the application, infrastructure, and DevOps teams—for example, security. Many datacenters use an intrusion prevention system (IPS). Your security team, if it applies the on-prem rules, will require an IPS running in the CSP that it can control. The problem is that IPSs tend to need access to layer 2 of the OSI Model, which most CSPs will not allow. Security (and other) teams need to rely less on specific tools and rather think at a higher level about what risks an IPS mitigates in the first place and how the CSP addresses those concerns.

Ultimately, your organization needs to be comfortable with the new roles and responsibilities, and with not having 100% control of the environment. Do not underestimate the difficulty in asking people to give up control to a CSP. Building a comfort level with workloads in CSPs takes careful planning, research, testing, and time.

And issues to do with lack of control or ownership extend far beyond just the security team. Examiners and sometimes customers in regulated industries previously had the right to audit an environment; while onerous, since you controlled the environment, you could comply. With CSPs, you typically do not get the right to audit. You might be able to get the right to examine, but that is fundamentally different. This may mean a change in customer contracts, and ensuring that your regulatory agencies are OK with the move (many now do accept CSPs).

Hopefully you can see that moving from on prem to CSP takes a lot of thought, coordination, and consideration. Using CSPs requires a learning curve and maturity. Take the time to do it right.

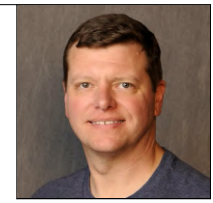
PART IV

Security and Compliance

Security at Cloud Native Speed

Chris Short

Principal Technical Marketing Manager at Red Hat



Cloud native technologies, like Kubernetes, are chosen by organizations to create a competitive advantage. Containers, service meshes, and serverless computing aim to jumpstart developer productivity. But they can change the attack surface of applications and infrastructure. We must protect cloud provider APIs, developer tooling, and applications. The good news is that we can use native Kubernetes tooling and cultural changes, like DevOps, to improve security postures and reduce the blast radius, all while improving developer productivity.

Struggles

Doing less with more is now the modus operandi for many organizations, which are moving toward cloud native infrastructure because they need to make improvements faster. Airbnb became a Marriott competitor almost overnight. Square and Stripe have changed the way ecommerce works. It's hard to imagine being an established organization and not worrying about market share.

People look to cloud native tooling to improve velocity, but the ecosystem contains a vast expanse of tools from many organizations. One glance at the Cloud Native Computing Foundation landscape tells the story: it has an overwhelming array of tools in categories like service discovery, CI/CD, storage, and many more.

Velocity

According to a recent industry report, between 2018 and 2019 "the number of containers that are alive for 10 seconds or less has doubled to 22%." If that doesn't move your needle, maybe this stat from page 23 will: 73% of all containers live for 30 minutes or less. That's right—a majority of containers run for less than 30 minutes. The market is responding to the increasing demand

for feature delivery. Consumers want more things faster than ever. We want more feedback, too. Rapid prototyping, DevOps, site reliability engineering (SRE), and other industry practices have brought us to this point.

Continuous Security

Continuous security should be a part of your pipeline. Securing software at speed requires automated processes. The last thing anyone wants to do is slow feature delivery. When security checks occur, they must be iterative and limited in scope. Gone are the days of security teams showing up at the end and holding up progress. Continuous security means security can't be a gate, but it can be a process ID (PID)—preferably, lots of PIDs throughout the pipeline and life cycle.

Platform Security

Securing platforms to reduce the blast radius is a vital part of cloud native security. Kubernetes is at the center of it all, but there is no silver bullet for securing Kubernetes. Security tooling in Kubernetes takes a layered approach down to the pod level. Kubernetes distributions provide role-based access control (RBAC), identity and access management (IAM), logging, auditing, and many other tools. Excellent platform security also uses security features that are parts of Kubernetes and Linux, such as:

SELinux
: Developed by the National Security Agency (NSA) in the early 2000s to secure Linux for government systems, SELinux has become a vital tool in defending against upstream Kubernetes vulnerabilities.

Secure computing mode (seccomp)
: This kernel feature, which filters system calls to the kernel from a container, prevents a container from executing calls not already on a predefined list.

Control groups (cgroups)
: Control groups manage system resource usage—CPU, memory, disk I/O, etc.

Security policies
: Policies can codify business requirements and apply them cluster-wide.

Although not a security feature, *namespaces* divide cluster resources among users, reducing the blast radius.

Practitioners cannot forget that the security landscape is ever-changing. They should keep an eye on the Open Web Application Security Project (OWASP) Top 10. Using static analysis of code at rest and dependency scanning tools is encouraged. The security posture can be improved by pulling trusted base images from trusted registries.

Speed Makes Us Safer

Safety can no longer be slow, methodical, and checklist-based. Concepts like DevOps have challenged assumptions about safety in software, and shown that automating our security and safety features brings many advantages. When security is automated, it runs at the same velocity as software development teams. Teams can test the resilience of their systems and continue to harden them as part of feature delivery. This results in less downtime, faster average recovery times, and speedier feature delivery to our customers.

Essentials of Modern Cloud Governance

Derek Martin

Principal Program Manager for Microsoft Patterns and Practices

There are four essential elements to consider when developing a governance structure for your cloud journey. Failure to address these points frequently leads to a variety of pains that are difficult to undo. These four elements are as follows:

- Subscriptions matter.
- The network has to come first.
- Security is essential.
- Automation is required.

Subscriptions Matter

The fundamental container of resources in Azure is the subscription. How many subscriptions do you need? Start with three and grow beyond that based on these conditions:

- Subscription capacity is exhausted.
- Acquisition and ownership (not just management) of Azure resources takes place in multiple geographical/political/regulatory jurisdictions.
- The "thing" being deployed to Azure is part of your company's "cost of goods sold."

This works for most companies. The first subscription is Production, where no standing security access exists (except for your CI/CD runners) outside of Reader roles. The second is Not Production. This subscription is where coordinated nonproduction tiers exist (Dev, Test, Int, Stage, PreProd), with an increasing security posture as the tier level approaches production. The third is your Hub subscription, where core networking, ExpressRoute circuits, etc.

are housed and heavily restricted. Visual Studio subscriptions should be provided to developers and IT pros for them to learn and do playground-based work. Keep a tight policy lock on these to prevent data exfiltration. When the developer or IT pro is ready to integrate with others, they move into the controlled Not Production subscription.

The Network Has to Come First

You cannot govern the cloud without a stable network topology. No amount of serverless or PaaSification of your environment eliminates the need for proper design, operation, and control of networking. These designs tell your application how to operate securely, fail over, and survive a data loss. Ignoring the network is a tragic and expensive mistake. You need not use hub-and-spoke routing and record all traffic. Other solutions exist that better lend themselves to modern network security and intrusion/breach prevention than forced tunneling, like Azure Security Center, Monitor, and Advisor for detailed, live introspection on what is happening in your environment in a correlated manner.

Security Is Essential

Implementing least privilege access and regular account reviews is essential. RBAC, principally applied at the resource group level via automation with zero standing access to production, helps prevent a whole host of unwanted experiences. Adopting an "assume breach" stance for everything allows you to focus your data protection efforts where they matter most: in the source system. Subscription-level access should be limited to your automation accounts, break-glass account, and audit solutions (read-only). Privileged identity management and multifactor authentication (MFA) for sensitive operations should be the norm. The need to "see" things via the portal or CLI should diminish the closer you get to production. For example, it is not necessarily true that the SQL administrators need full permissions to the resource groups where the SQL servers are located. Perhaps they need only Reader rights, or no rights except to the emitted logs. You should *never* allow a change to production absent automation. If you do, your disaster recovery strategy is invalid.

Automation Is Required

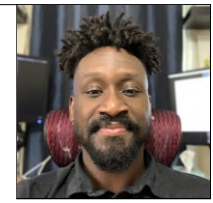
You cannot effectively manage the cloud via the portal. You cannot effectively govern or secure the cloud without automation. There are simply too many moving parts, too many places to make a mistake, and far too many

neat little buttons to push. Starting in your development tier, teams should have portal access to help craft automation scripts that include not just the application, but also the infrastructure *and* the configuration. As you move closer to production, the rights should be reduced at each tier until nothing but access to the emitted logs remains. This final step is *hard* and represents an ongoing journey. Reasons will always exist for deviation from automation, but those should be backported to your CI/CD pipelines to return your environment to an automated state for deployments, monitoring, and recovery.

Know Where the Secrets Are Kept and How

Emmanuel Apau

CTO, Mechanicode.io and Cofounder, Black Code Collective



The first order of business for a cloud engineer is ensuring you have a thorough understanding of where and how the secrets are secured. This knowledge will allow you to build an environment founded in security and provide protection from future accidents.

Let's answer the obvious question: what is a secret? *Secrets* are the organization's sensitive data; for example, passwords, certificates, application credentials, and API keys. I categorize secrets into two buckets: user and infrastructure/application. Each requires a different management workflow.

So what is secret management? *Secret management* is the central mechanism used to secure this sensitive data. There are three workflows to take into consideration when determining management methodology.

How Do We Share Secrets Between the Infrastructure and the Applications?

As cloud services and microservices proliferate exponentially, so does the need to ensure they can communicate with each other securely. Most cloud providers provide a means to manage this. However, it's the responsibility of the engineers to build the organization's conventions as a wrapper to these SaaS options. This wrapper could be an internal web app, a command-line tool, or checks during a CI phase that cover the following:

- The environment the secret exists for (development, staging, production, or global)
- The secret name
- The description
- Whether the secret needs to be rotated, and how often

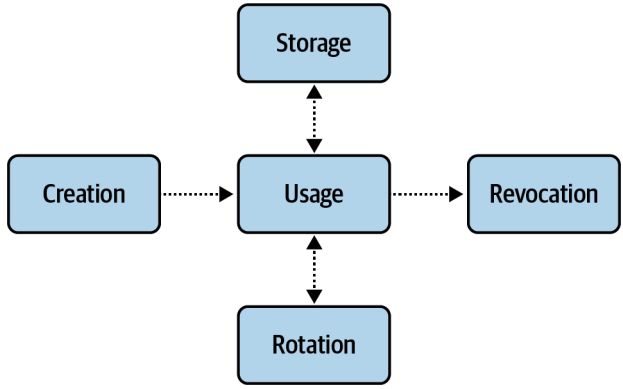
- Whether the secret has an expiration time to live (TTL)

The TTL will typically be double the rotation time, if one exists, to account for rolling back in case of disaster:

Environment	Name	Description	Rotation TTL	Expiration TTL
Production	mysql_user	MySQL admin user	60 days	120 days

How Do We Audit Our Secrets?

A cloud engineer needs to understand how change management comes into play for a secret.



A secret triggers different events as it passes through its life cycle. These events should cover which user/application caused the following to change:

- Creation date
- Modified date
- Expiration date
- Version number

Some level of versioning should exist so that it's possible to roll back to a previous state in case of a disaster.

Hooking into this event life cycle is imperative in keeping us aware of critical events such as secret expiration. These alerts will save team members a lot of sleepless nights, and provide them with more ownership over the ecosystem.

How Do We Share Secrets Among Users?

Sharing secrets among users is a simple concept that has led to many security issues, as this requires human intervention. The shortest paths to sharing secrets—email, instant messaging, hardcoding them into source code, etc.—have led to many unintentional security leaks.

You will want a solution that accounts for encrypting the data in flight to ensure secure transmission. Many SaaS options are available that account for this already, but when SaaS is not an option, password managers that are stored on the team's shared drives can be accessed by a shared primary password.

This user workflow is one that cannot be automated away and should be a priority for an organization to have documented and included in employee onboarding and security training.

Best Practices

So now that you are aware of your duties as a cloud engineer, you should consider a few points about secret storage, for whichever solution your organization decides to go with:

Rotate encryption keys
: Ensure that the cryptographic keys used to protect the organization's sensitive data are changed periodically to ensure security.

Identify management
: Determine who in the organization can manage secrets and to what degree.

Ensure encryption
: Make sure secret data is encrypted at rest and in flight during retrieval requests via an API.

Don’t SSH into Production

Fernando Duran

DevOps Team Lead at Kira Systems

Routine server system administration tasks should be handled with automation and services, through code and software. Not logging in to system consoles for manual routine maintenance can be seen as an indicator of capability maturity.

Logins to critical servers via SSH should be audited to determine who accessed the servers and what they did. Auditing can get complex when accessing servers via SSH is the standard policy and when considering cases like SSH forwarding and tunneling.

As a test, before logging in to a server to carry out a task, ask yourself the following:

- Was this task tested first in a dev/QA/test environment?
- Is this a one-off task (versus a routine task or request)?

If you answer no to either question, you should reconsider your workflow and think of ways to automate away the kind of work you SSH for.

Let’s review some common reasons a cloud engineer would want to log in to a server:

- To examine logs, like application, container, or operating system logs. This is a solved problem. Using a stack like Elasticsearch, Fluentd, and Kibana, or a third-party logging service in the cloud, will provide log aggregation, search, visualization, and permanent storage capabilities, with a proper life cycle and backups.
- For monitoring, to look at server telemetry like CPU/RAM/disk usage or exposed application performance metrics. This is also a solved problem; we have a myriad of commercial and open source tools at our disposition.

- For routine changes in the system, such as making configuration changes, patching the operating system, managing software installations and upgrades, and performing backups and restores. All these changes should ideally be done using infrastructure as code. We declare in code (which we keep versioned) our infrastructure and make our changes in code. Then, depending on our workflow, philosophy, and tooling, we can use configuration management tools, or we can re-create the server image, or we can use our favorite coding language and take advantage of the cloud vendor's software development kit (SDK) or API.
- Running tests. "Testing" in production can be needed to get a real view of application behavior; fake test data rarely behaves like the real thing. Or we may need to run a query that is not shown in a reporting server. While these are valid tasks, we should still avoid ad hoc manual operations and look into replacing them with code and systems that will perform such operations with less risk.
- "My server is a snowflake that needs constant TLC." Look into "cattle versus pets," because you have some problems.
- "I don't know what is running on this server or what this server is supposed to run." You have bigger problems you need to address.

There are a few valid reasons to SSH into a production server that is part of an application running in the cloud. Sometimes while troubleshooting, we need to log in to a server as a last measure because the information we have from the log and metrics servers is not enough to determine the cause of a problem. For example, we may not be getting logs or metrics themselves, or we may have network issues of the type "this host doesn't seem to be able to talk to this other host" and we want to verify that connectivity. We may also have hard Linux kernel issues, or strange behavior not explained by logs or indirect information. Another reason to SSH into servers is for the purpose of exploration or learning for new people in a team.

In any case, the next time you are about to log in to a server, stop and think: "How could I accomplish this task without manually getting into the server?"

Identity and Access Management in Cloud Computing

Isuru J. Ranawaka

Senior Full-Stack Cloud Developer

Cloud computing provides a shared network, computing capacity, memory, and storage on demand for a vast set of concurrent consumers. Concurrent access to shared resources has increased security loopholes and threats to the services running on cloud resources. Hence, *identity and access management* (IAM) is an imperative requirement in cloud computing. A cloud engineer should impose high application-level and network-level security compared to archaic approaches to avoid threats such as phishing attacks, denial-of-service attacks, and man-in-the-middle attacks. This article describes popular IAM patterns used in cloud computing.

Cloud-based clusters are primarily categorized into public clouds, private clouds, and hybrid clouds. *Public clouds* are offered by third-party providers over the public internet, making them available to any interested parties. *Private clouds* are isolated from public clouds and are operated on more secure private networks. *Hybrid clouds* combine private and public clouds. Furthermore, cloud-based clusters are exposed through different models, such as infrastructure as a service (IaaS), platform as a service (PaaS), software as a service (SaaS), and serverless.

Nowadays, a wide variety of free and commercial applications use IaaS, PaaS, SaaS, and serverless models. For instance, science gateways have been developed to enable research communities to easily manage and run their experiments on high-performance computing (HPC) systems. Those gateways are integrated into middleware that connects to the HPC cluster and manages experiments on behalf of gateway users. This multitenant middleware should be capable of providing seamless access to HPC resources for end users. HPC clusters are widely deployed on private clouds with high security privileges. Hence, accessing HPC resources requires specific SSH keys,

certificates, and password credentials. Thus, credential management, authentication and authorization, resource sharing, and access control are trivial requirements.

Furthermore, consider an application development platform that exposes APIs to develop retail support applications such as online shopping and data analytics apps. This platform should connect to Salesforce services, Shopify services, Google services, and Facebook services. End users create accounts on the platform and don't know of the existence of backend cloud services. Hence, the middleware should be able to successfully orchestrate cloud services, manage access credentials, and handle API authorization for cloud services.

By considering the aforementioned use cases and widely used industry practices, we can identify the basic elements of IAM as identities and groups, relationships, credentials, and entitlements. User accounts, user groups, user claims, user attributes, and user roles are amalgamated as identities and groups. Relationships describe the dependencies between identities, groups, attributes, and roles. Credentials represent the access keys of users and resources. Entitlements describe the access policies for users, groups, credentials, and relationships. Role-based access control, group-based access control, attribute-based access control, and policy-based access control are widely used access control principles.

IAM patterns are articulated from the aforementioned IAM elements. Delegated identity management (DIM), federated identity management (FIM), sharing, and synchronization are the most popular IAM patterns. The DIM pattern employs identity brokers to connect with external identity providers (IdPs), and identity brokers are responsible for just-in-time (JIT) provisioning. FIM agrees to trust different domains to allow applications to consume services across domains via a single user identity. Identity federation is categorized into *inbound* identity federation and *outbound* identity federation. Inbound identity federation allows external users to consume internal services, while outbound identity federation allows internal users to consume external applications. Sharing enables the common use of elements of IAM for services and applications. However, sharing may have limitations if user stores and applications reside in different domains. Synchronization is used to replicate data stores and application stores among different services.

Numerous technologies and protocols are used to implement the aforementioned IAM patterns. OAuth 2.0 is an industry-standard protocol for authorization. This simplifies client development and integration with identity servers. OAuth 2.0 is built on elements such as scopes, grant types, and client

types. Scopes limit access to user identities for applications. Grant types are OAuth flows of user authentication; widely used grant types include Authorization Code, Client Credentials, Resource Owner Password, and Refresh Token. OpenID Authentication 2.0 is an authentication federation protocol that relying parties can use to obtain verified identities from IdPs. OpenID Connect is an extension of OAuth 2.0 that is used by clients to obtain users' account attributes via claims. The System for Cross-domain Identity Management (SCIM) protocol is used to synchronize user stores residing in different domains.

Using IAM elements, IAM patterns, and integration protocols, numerous IAM cloud solutions are developed and consumed by a wide range of services and applications. Single sign-on (SSO), shared logins, service accounts, multifactor authentication, and identity linking are the most popular identity solutions provided by vendors in the security industry.

Treat Your Cloud Environment as if It Were On Premises

Iyana Garry

Cloud Penetration Tester

There is a saying in the industry that goes, "The cloud is just someone else's computer."

However, if you're storing sensitive information and user credentials in a cloud environment, should you treat that environment as if it were just someone else's computer? Cloud engineers manage data in the cloud, but what could happen if this data is unprotected?

A 2019 study by the security research firm Proofpoint found that 92% of Fortune 500 companies' cloud accounts had been attacked. These attacks could have been avoided if the companies had taken the time to harden their cloud environments as if they were on premises.

The following is a list of precautions to take in order to ensure that you're managing your cloud environment in a secure way:

- When configuring cloud infrastructure, one of the first lines of defense is encryption of at-rest and in-transit data. An SSL/TLS certificate and user credentials should be generated to enable HTTPS and SSH traffic. In addition, a firewall should be configured with inbound rules for that traffic, and each cloud server should have its own set of user credentials.
- Check that cloud account credentials are not hardcoded into any software. This includes version control repositories, public and private. Also, user credentials should be changed frequently. Your cloud platform may have a key management service that centrally stores and rotates the credentials for your servers.
- Each network service (the web server, database server, email server, etc.) should be run on its own instance. This makes it difficult for attackers to access all of the assets.

- As burdensome as it may be, use two-factor authentication (2FA) whenever you log in. Unless an attacker also has access to your mobile device, this can help prevent them from infiltrating your cloud account.
- Change the default configuration files and port numbers for your services. If you are using default configurations, your environment is at risk because attackers will know where to look once they are able to infiltrate with your user credentials. Give them a hard time and leave that */var/www/html* directory empty.
- Use your platform's monitoring and logging service to notify you when suspicious activity and/or traffic is detected and to log as many assets as you can: your servers, cloud virtual private network (VPN), file storage system, etc. If an attack is implemented in your environment, logging helps you determine which assets are being targeted.
- Although it may not seem like it, maintaining availability of your resources is a security practice. An attacker may not aim to infiltrate your environment, but prevent legitimate web traffic from flowing through it by flooding it with a distributed denial-of-service (DDoS) attack. Configuring a load balancer and a web application firewall (WAF) in your environment can help mitigate such attacks.
- Finally, back up your cloud assets frequently. In the event that an attacker infiltrates your account and wipes all your data (or possibly worse, if you or someone in your organization unintentionally wipes all the data), you need a copy to restore all of it. Whether you store your backups on removable media or use your platform's backup service, if one is available, this step is a vital part of securing your data.

Of course, you don't have to follow any of this advice. You may continue to make cloud penetration testers' jobs like mine easier instead. :)

You Can't Get Information Security Right Without Getting Identity Right

Sarah Cecchetti

Principal Product Manager, AWS Identity, and Cofounder, IDPro

Security is Job Zero for every cloud engineer. It is critical to remember that in a cloud environment, you are building for multiple tenants who should never be able to access or change one another's data. Every permission has to be perfect every time. The subfield of information security that handles people and permissions is called identity and access management—and it's a fascinating field!

The place where you store users' information is called a *directory* or *identity store*. If you don't have existing users, their information will be collected when they sign up with your application, and stored in your directory. If you do have existing users, you may want to consider a *federation model*. Such a model uses existing credentials (usually a username and password) to log users in, so that they don't have to go through the hassle of creating a new account. The most popular federation standards are Security Assertion Markup Language (SAML) and OpenID Connect (OIDC).

The most secure way to have users log in is through *multifactor authentication*. For this you want the user's credentials to cover two of these three categories: something they know (like a password), something they have (like a phone), or something they are (like a fingerprint). When I log in, I use a cryptographic token called a YubiKey. I wear mine as an earring so I never lose it! One of the most common ways to have a user confirm something they have is to send them a text message. While this method of multifactor authentication is a breeze to deploy, it's not very secure. Text messages are sniffable by off-the-shelf hardware and software, and they are vulnerable to SIM-swapping attacks that can allow an attacker to gain access to a valid

user's phone number. Many other ways exist to have a user confirm something they have, including authenticator phone apps, USB security keys, and one-time-password devices.

Once a user is logged in, they often need to delegate authorization to different applications in their tenant environment. For instance, they might want to authorize a travel application to access their name and passport number. It is tempting as a cloud engineer to allow an application to use the username and password of a user in order to access the user's data. Resist this temptation! Delegated authorization is most safely done through an identity standard called OAuth 2.0. You can easily build an OAuth 2.0 token endpoint into your application that will mint access tokens for applications so that they can access APIs on behalf of the user. These tokens are limited in what they can access, and for how long, so that the user's data is protected and their password isn't being passed between applications.

Once a user has completed their tasks, it's important to log them out and revoke any access tokens that are no longer needed. How soon you log out users depends on your application. If you have an application that doesn't deal with sensitive user data, you may want to keep users logged in for days or even months. If your application handles data of a personal nature, like medical or financial data, you may want to log users out as soon as they close the application.

Whatever identity experience you build, remember that it's the front door to your application. It needs to be both welcoming and secure. Having a terrible identity experience is especially bad for users, because proving their identity isn't what they're trying to do in the first place. If they are frustrated before they even get to the application, they are likely to have a bad overall experience. Make sure that you've architected and tested your identity experience so that it's simple, easy, and safe for your users.

Why Are Good AWS Security Policies So Difficult?

Stephen Kuenzli

Founder of k9 Security

> *Why is AWS IAM so @!#^$ hard?*
>
> —Every cloud engineer

Short answer: first because the powerful AWS security model is complex, and second because modern application deployments change rapidly. Let's examine why configuring *good* AWS security policies is so difficult.

The AWS Security Model Is Powerful but Complex

In the cloud, capabilities are delivered using services configured via APIs. Security capabilities are no exception. AWS security APIs enable customers to fulfill their security responsibilities within the AWS shared responsibility model.

Customers control access to their cloud resources and data by configuring security policies in the AWS security services. These security services evaluate the policies to allow or deny access. They include the organization's IAM services, and more than 20 data services that support resource policies.

Five types of AWS security policies determine whether an API action will be allowed: Service Control, Identity and Access Management, Resource, Boundary, and Session. Expert users of these security services can create robust access controls.

But this large set of security services, resources, and policy language is complex, difficult to understand, and hard to test without breaking things. Engineers need to understand and configure multiple policies to protect data. For S3, engineers must usually configure IAM policies, an S3 bucket resource

policy, and service-specific access controls like S3 access control lists (ACLs) and public access configurations.

How Policies Are Evaluated

The core concept of the AWS security model is as follows:

> Security policies associated with a principal or cloud resource control how a principal may interact with that resource.

A *principal* could be an IAM user, an AWS account, or an unauthenticated person from the public internet. *Policies* grant principals permission to access a resource. Policies are attached to the principal or the resource, such as an S3 bucket.

Consider applying the principle of least privilege, a security best practice. Engineers create a role for each application component and an IAM policy that allows the application to execute *only* the AWS API actions needed by the application.

Good news: principals are granted no access by default, and you can build access incrementally. Bad news: constructing least privilege IAM and resource policies by hand is difficult. AWS uses all the types of security policies during the access decision-making process. It's easy to get something wrong, even for experts.

Did you notice there are *two* paths for accessing a resource that supports resource policies? Look for the green end states.

Both a resource policy attached to a bucket and an IAM policy attached to an IAM principal may grant access to an S3 bucket. If *either* the bucket or attached IAM policy allows access to the bucket, the IAM principal is granted access.

To grant least privilege, engineers must carefully engineer conditional access in policies:

- IAM policies should limit which resources a principal may access using resource conditions; e.g., allow the *firewall* role to access the *logs* bucket, not all buckets.
- Resource policies should *allow* intended principals and *deny* everyone else; e.g., allow the *credit-processor* role to access the *credit-applications* bucket and deny everyone else.

And then get ready for change.

Cloud Deployments Change Rapidly

The number of identities used to manage and operate technology services is growing rapidly. Both business and technology trends drive this growth.

First, successful organizations grow. Second, organizations are decomposing application architectures to scale application ownership across teams. Third, organizations that deliver application changes quickly and safely to customers have a competitive advantage.

This growth and change is good for the organization, but stresses engineers writing and reviewing policies without support.

Summary

Cloud security policies are complex and difficult to validate. Cloud deployments' identities and resources are growing and changing quickly. Adopt practices that simplify the way practitioners understand and secure cloud deployments and integrate with automated delivery processes to protect data.

Side Channels and Covert Communications in Cloud Environments

Will Deane

Director at ASX Consulting Ltd

Side-channel attacks abuse information leaked by the processing system, rather than directly attacking the system itself. Attackable side channels include analysis of power, electromagnetic emissions, acoustics, heat, and timing. Historically, side-channel attacks were predominantly focused on cryptographic systems. With the adoption of hypervisors and cloud computing, recent research has focused on cross–virtual machine side channels, mostly using CPU cache timing techniques. Even though side-channel attacks are typically slow and often provide only partial data recovery, sophisticated attacks have been demonstrated in public clouds, including stealing encryption keys and creating covert channels between cooperating non-networked systems.

In 2009, researchers from the University of California and the Massachusetts Institute of Technology published a paper demonstrating techniques for coresiding an attacker's virtual machine on the same physical host as a victim in Amazon EC2.[1] They also demonstrated some basic side-channel attacks, including low-bandwidth covert channels between cooperating coresident hosts using both hard disk and memory bus contention timing. By 2017, researchers from Graz University of Technology in Austria had developed a practical covert channel providing 45 KBps of bandwidth using CPU cache timing.[2] They implemented a Transmission Control Protocol (TCP)

1 Thomas Ristenpart et al., "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds," *Proceedings of the 16th ACM Conference on Computer and Communications Security* (November 2009): 199–212, *https://doi.org/10.1145/1653662.1653687*.

2 Clementine Maurice et al., "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud," Network and Distributed System Security Symposium (February 2017), *https://doi.org/10.14722/ndss.2017.23294*.

stack on top of this channel and demonstrated streaming a music video across the channel at Black Hat Asia 2017.

So, what do those attacks mean for the use of public cloud infrastructure? For covert channels, an attacker has to get their code running on the victim virtual machine and also get their VM coresident on the same physical host. Lots of methods can be used to get malware onto a running system, but for typical cloud workloads, common techniques include the following:

- Using stolen credentials
- Phishing developers or DevOps
- Attacking supply chains
- Compromising CI/CD pipelines

The coresidence requirements can be met by brute-forcing virtual machine creation and testing for coresidence with the victim; this is made easier if the attacker's code is already running on the victim VM.

These requirements suggest that cloud users processing sensitive data on virtual machines without direct internet connectivity who are subjected to threats from well-funded, sophisticated threat actors are most at risk from these attacks.

For traditional side-channel attacks, cryptographic keys are most at risk; for example, keys associated with Transport Layer Security (TLS) services. These attacks don't need software running on the victim machine; however, this makes confirming coresidency more difficult. These attacks are also prone to noise-based errors, and at the time of writing, I'm not aware of any practical demonstrations of real-world key recovery from realistic public cloud workloads.

It's said the NSA has a saying: "Attacks always get better; they never get worse." Research into side-channel attacks seems to follow this axiom. In January 2018 details of the Spectre and Meltdown vulnerabilities affecting CPUs were published, with further related vulnerabilities published throughout 2018, including Spectre-NG, ret2spec, SpectreRSB, and NetSpectre, to name a few. This trend continued in 2019 with related attacks published including Fallout, RIDL, ZombieLoad, and Spectre SWAPGS.

Although many of these issues can be remediated with software patches, some (such as cache timing attacks) abuse underlying hardware operations and require hardware architecture changes to mitigate—something that isn't going to happen in the short to medium term.

If you are running sensitive workloads at risk of side-channel attacks in public clouds, all of the major IaaS suppliers provide options to use hardware dedicated to you, removing the option for an attacker to become coresident with your virtual machines. AWS offers Dedicated Instances, Azure has Isolated Instances and recently announced Dedicated Hosts, and Google Cloud has sole-tenant nodes. These options all cost a little more than their standard shared tenancy offerings, typically in the range of 6% to 10% for pay-as-you-go pricing.

Organizations should consider the risks of side-channel and covert communication attacks to their public cloud–hosted systems and decide whether to accept the risks or pay the additional cost for sole-tenancy hardware.

PART V

Operations and Reliability

When in Doubt, Test It Out

Dan Moore

Principal at Moore Consulting

When I taught AWS certification courses, I'd often get questions about how a service behaved under load or other unusual circumstances. Frequently, I could answer from personal experience or by asking other instructors; occasionally, class members provided their insights. Sometimes I could dig up relevant vendor documentation.

However, my default answer was, "Test it for yourself. There's no substitute for testing."

This is one of the great advantages of the cloud. When you have a question about the performance or behavior of a service or system, spin it up and test it. This will cost you money and time configuring the system, but certainly will be cheaper than ordering hardware, racking it, and then also configuring the system. When you're done with your testing, you can tear down the infrastructure and never worry about it again. This sure beats shipping a server back to the manufacturer.

Of course, no testing scenario can replicate production perfectly. But you can get pretty close (especially if you can reuse production traffic).

When you do test, start by documenting what you want to achieve. What is the question you are trying to answer? Make sure to seek feedback from other team members and/or search online, as it's possible someone has already answered your question. If you do find answers, understand under what circumstances the tests were performed, as the cloud and the offered services change over time.

Here are some examples of cloud infrastructure questions you might want to answer:

- How do Elastic Block Service (EBS) volumes of different sizes and types perform under load?

- When a Kubernetes cluster running on Google Kubernetes Engine (GKE) is under load, what happens when you add an additional node? An additional pod?
- What happens when you turn off a network address translation (NAT) gateway while a file is being uploaded to S3 from an EC2 instance in a private subnet (without an S3 VPC endpoint)?
- What is the cold start time for an empty Azure function? What about a function loading your DLLs?

Think about what steps you are going to take to try to answer the question.

With your question and methodology spelled out, spin up your testing environment. Having your infrastructure represented as code will make this faster, especially if you have a complicated environment. If you are creating the test environment manually, record settings and other configuration in a text file so you can re-create the environment later.

Run your tests. If you are load testing, find an open source or commercial load-testing tool. What you need depends on your goals: you need a different tool to test 100,000+ simultaneous users on a website than you do when trying to understand how an internal API handles 100 requests per second.

Review the data to see if it answers your question. More questions or areas of interest may appear. Adjust your tests to answer them.

Once you have your answers to the desired level of certainty, tear down your testing infrastructure. Document what you tested, how you tested, and your results. Circulate this internally to help your team. If possible, publish it on your company blog to both help others in the same boat and boost your company's standing in the community. All the vendor documentation in the world is no substitute for rolling up your sleeves and testing.

Never Take a Single Region Dependency

Derek Martin

Principal Program Manager for Microsoft Patterns and Practices

Enterprises of all shapes and sizes frequently make a tragic mistake when migrating to the cloud: "Hooray, I no longer have to have a disaster response or business continuity plan; Microsoft/AWS/Google will take care of it!" It's true, the cloud is a highly resilient place to migrate your enterprise to; but at its most basic, even the cloud is composed of physical servers and run by human beings. Things break, catch fire, go boom, and are occasionally susceptible to human error. So, when migrating to the cloud, take this first and most important lesson with you: never take a single region dependency!

Azure provides a variety of solutions to help you be successful, including these:

Redundant storage
: Keep multiple copies of your data in multiple physical locations.

Availability zones
: Run your solutions in multiple data halls/facilities in a single region.

Backup
: Enable file-level restorations for when files go missing.

Site recovery
: Fail over VMs from one region to another with a very tight recovery time objective/recovery point objective (RTO/RPO).

Each stock-keeping unit (SKU) within Azure takes a slightly different approach to regional resiliency, and your enterprise needs to consider those methodologies for each of your applications. For example, Azure SQL Database can automatically regionally fail over in the event of a disaster. Cosmos DB distributes reads and writes globally without additional configuration.

Consider the service-level agreement (SLA) of each Azure SKU in your application as well as its regional resiliency plans. Then work backward to your final design. Here are some major points to keep in mind:

- Always have a break-glass account ready; this should be a cloud-only identity without multifactor identification and locked in a vault.
- Network failovers should be automatic. Leverage zone-aware infrastructure and global endpoint routing, offered by services like Traffic Manager and Front Door. Have network routes that can easily adjust based on a variety of outage conditions.
- Practice your failover—the best-prepared companies do *not* rely only on tabletop exercises. Fail over your environment frequently, and do it in production. Site recovery can help with your VM failover and failback.
- Your application must support multiregion awareness. Typically, this isn't hard for the frontend, but the data tiers will require careful consideration of the RTO/RPO.
- No two regions are exactly alike. This is by design. Careful planning is required to make sure the SKUs you need are available in your chosen failure region.
- If you are *not* relying on geo-redundant storage (GRS) as part of your disaster recovery strategy, *do not* use the peered region as your failover. In the event of a disaster, the peered region will be quite constrained. If you aren't using GRS (which requires you to fail to the paired region), fail out somewhere else!
- A resource group has a region assignment. Resources inside the resource group need not be in the same region. Control plane operations of all resources in that resource group happen in the region to which it is assigned. In the event of a region failure, the resources inside that resource group will not be able to be controlled, even if those resources are in a different region (although they will be running just fine).

The most resilient organizations run applications and systems in multiple regions in production. In the event of a regional failure, some applications may be degraded because they fail over to another region running smaller or less expensive SKUs. Other applications will remain healthy. These organizations leverage Azure Virtual Network (VNet) peering and traditional hub/spoke or full mesh connectivity to make the entire organization as regionally independent as possible. They practice. They automate. They audit. They survive!

Test Your Infrastructure with Game Days

Fernando Duran

DevOps Team Lead at Kira Systems

"You don't have a backup until you have performed a restore" is a good aphorism, and in a similar way, we can say that your service or infrastructure is not fully resilient if you haven't tried breaking it and recovering.

A *game day* is a planned rehearsal exercise in which a team tries to recover from an incident. It tests your readiness and reliability in the face of an emergency in a production environment.

The motivation is for the teams and the code to be ready when incidents occur; therefore, you want the test incident to resemble a real-life incident. When you run these experiments in production environments and in an automated way, this is called *chaos engineering*.

There are several items to consider when preparing for a game day. Most importantly, you need to decide whether you are running the exercise in production. This is the ideal, since any staging or test environments are never really going to be the same as production. But on the other hand, you have to comply with your SLAs, obtain approval, and warn customers if needed. If you have never done a game day or if the target system has never been tested for disruption, then start with a test environment.

Another decision is whether you want the procedure to be planned and triggered by an adversarial *red team*—in this case, one team or person very familiar with the system will create the failure without warning.

You'll want to run these game days periodically (every four or six months, for example), as well as after a new service or new infrastructure has been added. This time frame may also depend on the recent history of past responses to real incidents. An incident exercise should run for a few hours at most; you don't want long-lived lingering effects.

Different types of failure can be introduced at different layers:

- Server resources (such as high CPU and memory usage)
- Application (for example, processes being killed)
- Network (unreliable networking or network traffic degradation: adding latency, packet loss, blocked communication, DNS failures)

Gray failures (degradation of services) are often worse than complete crashes, since the latter have a short feedback loop. Also, degradation can be hard to produce.

Before running a game day, you need to determine the following:

- The failure scenario or scenarios
- The scope of systems affected and what can go wrong (having a contained "blast radius")
- The *condition of victory*, or acceptance criteria for the system to be considered "fixed"
- The time window for recovery—estimate the duration and add 2× or 3× just in case
- The date and time
- Whether you are warning beforehand
- The team or people on call at the time who will handle the incident

You also need to prepare the response team(s) and how they are going to work. A common approach to incident management is to have one person focused on solving the problem surrounded by supporting people or teams so that person doesn't have to worry about anything else. For communications, a chat channel is better than email or phone since it works in real time, allows multiple people to collaborate, and leaves a written log.

The incident manager—the person leading the response team—doesn't have to be a manager or the person who is most familiar with the system. Indeed, it's better for knowledge dissemination that it be a different person. Besides, you want to make sure you don't have a "bus factor" of one. If there are runbooks to recover from an incident, this is also a way to test such documentation, by having someone different from the author going through it.

You may want to also have a coordinator to answer to business units and executives; you don't want them asking for updates and distracting the incident manager. Other teams can be observers; game day should be a learning opportunity for all.

During game day, document while the incident is ongoing, with timestamps, observations, and actions taken. After game day, perform a postmortem to answer questions like these:

- What did we do and how can we do better?
- Did the monitoring tools alert correctly in the first place, and were those alerts routed by the pager system to the person or teams on call?
- Did the incident team have enough information from the monitoring, logging, and metrics systems?
- Did the incident team make use of documentation like playbooks and checklists?
- Did the members of the incident team collaborate well?

If needed, update the technical documentation and procedures, and disseminate the lessons learned.

Improve Your Monitoring with Visualizations and Dashboards

Jason Katzer

Consultant and Creator of CloudPro.app

Charts take your monitoring metrics to the next level by making it possible to visualize them.[1] Why work on this task? Because viewing the charts you create will unlock a different kind of creativity when facing an impending incident in your production systems.

It may take some time to get this right; creating effective charts that tell you what you need to know about your application is more of an art than a science. For example, you can use login metrics to start to understand the current traffic flowing through a certain user path, while also being able to contextualize that instantly with the insight of overlaying the history over certain time periods.

Additionally, changing the aggregation you are using can tell you different things. Think back to calculus class, if you took it: the current value of a metric can tell you something interesting, but the rate of change of that metric can indicate a trend that tells you something different.

When you are having an argument about how often one thing happens versus another, that is a perfect time to turn to metrics. You can add a line of code to track something, ship it to production, and start to get answers instead of guessing.

Functions allow you to permute the data being graphed in a limited set of ways as made available by your cloud provider. Sometimes overlaying the same metric on the same graph multiple times, but applying different functions to each line, can give you even more information instantly about a key metric.

1 Excerpted from *Learning Serverless* by Jason Katzer (O'Reilly, 2020).

Some of the basic functions you should expect to have are sum, avg, min, and max. These functions are the building blocks to create more powerful visualizations that give you the most insight into the operation of your services and systems. Make sure you are familiar with all of the offerings of the tool you are using.

Documentation is a must for each graph you create. Explain what the human operator is looking at and how to make sense of it. Reference or link to additional information in your runbook or operations manual. And if there is a specific reason you chose certain functions to display data that wouldn't be easily understood by someone unfamiliar with the system, add that reasoning to your documentation.

Creating a dashboard of effective charts can bring what was once an invisible system to life with full visibility into all of the vital signs of health for your service or even the application as a whole.

Many dashboards have the ability to change the data displayed based on changing the value of a tag. You can reuse dashboards between the different stages of your services by utilizing this functionality.

Well-designed monitoring dashboards that are easy to understand and reflect the health of the service should allow you and others to detect anomalies and incidents. Here is where you are actually able to understand the health at a glance.

Generally, dashboards should allow you to adjust for the time period in which you would like to inspect. Including important events such as deploys can help a human operator determine whether an issue was directly caused by a deployment and investigate further.

No one dashboard will solve everything for everyone. You might combine multiple tools and sources into a dashboard. Either way, you'll need a consistent set of rules and standards for all of your "official" service dashboards.

REvisiting the Rs of SRE

J. Paul Reed

Senior Applied Resilience Engineer at Netflix

One of the hottest topics in site *reliability* engineering right now is how to make your applications and services *resilient* in the face of failure. And, as most site reliability engineers know, one of the main arguments for moving those services into the cloud is its much-touted *robustness*; but, of course, we must think differently about how we architect our applications if we expect them to "automagically" *rebound* when something goes amiss in the technological sky.

Engineers frequently run into these R-words in discussions on how to develop and operate in the cloud. Hearing them so often, you might have started to wonder: don't they all sorta…mean the same thing? Fear not: resilience engineering (RE) is here to help clarify all those Rs!

Resilience engineering has existed as a subdiscipline within the safety sciences for over two decades; practitioners recently started to apply its concepts to our industry, looking at how human factors, ergonomics, and "safety" relate to improving the functioning of the web-scale systems that developers and operations engineers wrangle with daily. A major point of examination is the contributions we messy humans make to our systems.

In resilience engineering, those R-words refer to specific (and different) aspects of the socio-technical systems within which we exist and maneuver:

Robustness
: RE defines robustness as "System *X* has property *Y* that is robust in sense *Z* to perturbation *W*." Put in less "mathy" terms: *robustness* is created when we design and implement fallbacks in microservices or use languages with, say, memory safety guarantees (Java instead of C++). Robustness protects our systems against a *specific* type of failure mode, and offers that protection in a *specifically designed* way. (When folks say their service is "resilient" to failure, RE would say it is "robust in the face of this specific set of failures.")

Reliability
: This R-word refers to patterns that support consistent operations or service levels; examples include making an application multiregion in the cloud or developing an application on two operating systems and fixing a bug that technically exists on both OSs, even if the application actually crashes on only one.

Rebound
: In an RE sense, *rebound* refers to the ability of a system to deal with a chaotic situation (an incident, say) using structures that have been developed and deployed *before* it's confronted with that chaos. During an incident, a CI/CD pipeline that encodes all the deployment logic (so nothing accidentally gets missed in the fray) is an example of rebound capabilities.

Resilience
: Last but certainly not least, RE defines *resilience* as a set of properties and practices that increase a system's ability to react to the world around it. The important distinction here: resilience is something you "do," not something you "have." "*Resilience* is a verb," it's often said. Specifically, there are activities, both "socio-" and "technical," that we as engineers can leverage to foster resilience, including establishing and sustaining common ground to increase inter-team predictability, setting the stage to cultivate personal and team adaptive capacity to leverage in future incidents, and facilitating company-wide organizational learning efforts.

Resilience engineering's foray into software development and operations offers a unique opportunity to bring rigor and hard-learned lessons from other industries—aviation and air traffic control, healthcare, maritime operations, construction, nuclear power, and more—to the increasingly critical, society-impacting systems we are responsible for.

We (self-described) "software safety nerds" invite you to check out resilience engineering as a way to improve not only the cloud-based applications you're responsible for, but your own experiences as you operate them.

And bonus: now you always know which R-word to pick when discussing how to make that pager quieter during your team's on-call shifts!

The Power of Vulnerability

Ken Broeren

Founder of Elevaros

What does it mean to be *vulnerable*? We rarely use this word because, for many of us, vulnerability equals weakness. A quick story…

It was a typical Thursday, and I returned to my desk from a meeting about a project I had been working on for months. Our team had been making good progress, but we still had a long way to go.

I scanned through my email—no major emergencies there—and got to work. A few minutes later, my phone vibrated itself off the desk from the flurry of text messages. Several "system down" errors and "Hey, what's going on with the network?" messages stared back at me.

"So much for getting any work done," I muttered. One look at the bright red network dashboard told me links were down everywhere. I started thinking of scenarios that could have caused such a widespread issue.

I opened an emergency bridge line and got the key people on to start troubleshooting.

"It could be our provider—is the telco having a major issue right now?" someone asked.

"Good question," I answered.

"Someone give their support a call and get a ticket opened right away," our director barked. Jeff volunteered to do that and dropped off the call.

"Did anyone make any changes?" the database engineer asked. "I can't connect to any of my production database instances, but I can get to a few of my development servers."

"No changes that I am aware of," and "The change control calendar has nothing on it for this morning."

We had a lot of back-and-forth discussion with no real progress. Finally, we made a breakthrough. All network traffic stopped at one of our internal firewalls.

One of the network engineers piped up, "Umm, this ruleset changed this morning—looks like 10:17 local time."

"Damn it, another unauthorized change," I thought. I immediately suspected one of my team members.

I was pretty sure John had made the change, so I wrote a message to him: "We're having a major outage. Made any changes to the firewall this morning?"

He didn't answer for what seemed like a long time, though in reality, it was 90 seconds. "Nope," he said. "Not me." "Who else could it be?" I muttered under my breath.

The network engineer said, "I think it was Mike." Mike was usually careful about making changes. Without chastising him, I asked, "Mike, can you get that change reversed right away?"

To my surprise, Mike piped up, "Yep, I'm doing that now."

Wait a minute, I thought. Was Mike on the call the whole time? Why hadn't he said anything?

Well, the truth is he was afraid of being vulnerable. He feared taking the blame for the outage and wanted to avoid taking the fall for the error. At first, he tried to convince himself that his "innocent" change couldn't have been the culprit. But as more evidence pointed toward his action, he had no choice but to confess.

But that's not vulnerability. Hiding from your mistakes is the opposite of being vulnerable.

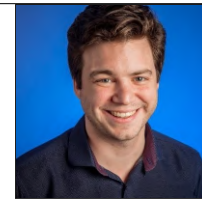
What if we all approached outages with a "Hey, that might have been me!" attitude? We all make mistakes, and we're not competing with one another, especially during an outage. Outages make us all look bad—not just the one who made the error. Being open and transparent solves issues quicker and prevents more outages!

Great teams don't hide things from one another. They collaborate and shoulder the load together.

Be courageous! Vulnerability isn't a weakness. It is a strength that you can bring to your team.

The Basics of Service-Level Objectives

Kit Merker,
Brian Singer,
and Alex Nauda
Nobl9


"I want it to work perfectly," your boss or maybe a product manager tells you. But you, as a cloud engineer, know they aren't really willing to pay for that level of service, even if it were possible, which it's not.

How can you give management an easy way to instantly understand the trade-offs between reliability, speed of innovation, and cost? Service-level objectives (SLOs) are the answer. SLOs create clear reliability guidelines that balance the trade-offs between cloud costs, speed of change, and external risks.

What Are SLOs?

SLOs are key performance indicators for cloud services *based on customer happiness*. SLOs define the precise level of service that needs to be achieved in order to avoid unacceptable risk of displeasing the customer.

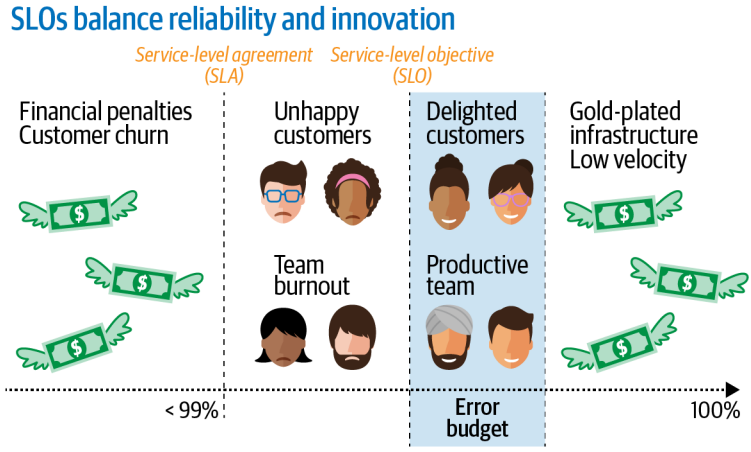
Let's use availability as an example. When we talk about how often our infrastructure is available (uptime), we typically speak in terms of *nines*. If your infrastructure is available *four nines*, or 99.99%, it will be unavailable 52.6 minutes a year. However, if your infrastructure achieves *five nines*, then your system is up and working 99.999% of the time—that is, it's down only 5.26 minutes a year.

In an ideal world, we'd want our infrastructure to achieve as many nines as possible; however, moving from one class of nines to the next-higher class is roughly 10 times more expensive (you'll incur significant people and infrastructure costs to make the leap to the next level). And when you consider the inherent limitations of physics and the architecture of public networks, approaching five nines of reliability consistently can become nearly impossible. So, the key question is, how many nines are good enough to keep the customer happy without wasting resources?

Another way of looking at SLOs is to consider their corresponding *error budget*, a small acceptable allowance for errors. If our SLO is 99.99% uptime, our error budget is the additive inverse: 1 – 99.99% = 0.01%, or 52.6 minutes a year. In other words, we will tolerate errors within this allowance—this small amount of downtime—because we expect that outages within this range will not upset customers enough to warrant prevention.

SLOs: The Cloud Engineer's Best Friend

SLOs help bring organizations together around reliability. You're not just chasing nines for nines' sake, pretending to be perfect, or hoping dumb luck is on your side.



Here are three ways SLOs can be a cloud engineer's best friend:

SLOs help you collect data about how well you are serving your customers.
: Achieve your SLOs, and customers remain happy; blow your error budgets, and customers leave.

SLOs enable you to evaluate business trade-offs.
: SLOs help you balance the competing interests of product stakeholders who want to rapidly launch new features or products, and IT operators who want to maximize infrastructure performance and reliability.

SLOs serve as a common language between business and multiple technical stakeholders.
: SLOs help everyone in the company work together as an aligned team to delight users and grow the business—and do it without guessing.

Where Do You Start?

As in the preceding example, availability of cloud services is a great place to start. You may also want SLOs to address quality, throughput, latency, and more. In fact, you will eventually have not one but many SLOs, because a variety of things to be measured play a key role in delivering customer happiness.

To get started, choose the cloud service aspect that is most critical to your business. Then, as your teams learn what SLOs are all about and what a great tool they are for balancing reliability, cost, and innovation, broaden your scope to include SLOs for other aspects of your cloud services, such as automation, infrastructure, applications, and user journeys.

As you expand the reach of SLOs in your organization, remember: SLOs must always be directly connected to customer happiness and business impact.

Oh, No: No Logs

Laura Santamaria

Developer Advocate at LogDNA

Logging is the least glamorous part of any server, much less any cloud-based system. However, it's one of the most critical processes on a server for operations teams, whether they're responding to an incident or monitoring a system for anomalies. Great logs are even more crucial for anyone working with a distributed, cloud-based system that has not been properly maintained—the dreaded legacy system. However, what do you do when you don't have logs?

If you don't have logs, don't panic. There are many ways to debug a system without logs. It just requires ingenuity and patience, along with creative thinking. If you panic before you can get too far, you will narrow your mental pathways and take up precious thinking space worrying about your boss breathing down your neck. Take a deep breath and sit down to think.

First, try hitting any available endpoints or services. Whether it's an API call through `curl` or a ping to a server, you will get something you can work with, even if it's no response at all. For example, a 403 response from an API with known good credentials indicates an error with authentication, perhaps from a compromised authentication system or someone updating a database without following a procedure your company has established. No response from a server or an API tells you to take a closer look at networking connections if you've already checked the cloud provider's user interface to ensure that the servers hosting the system overall are actually up.

Next, drill down into any available systems you can access. Getting direct SSH access to any boxes can be a huge boon, as you can check on the state of any running processes and see which processes might have shut down. Your system *will* be logging data on those boxes, and an SSH connection can allow you to see those logs. On continuously updating operating systems like CoreOS, it's possible that updates to the operating system broke an application, especially if legacy distributed systems are running. Directly accessing boxes will quickly bring those problems to light. You'll have access to the system logs, which can help correlate timing on any system changes to the

start of any incidents. Don't discount the problems a system update underneath your container-based systems or distributed systems can cause.

If you don't have SSH access or if it's a shared server—such as with a serverless system—you may need to take a closer look through a codebase or try accessing secondary backend systems like your databases to see if you can gather any data on the state of the system from there. Get creative. Think through the architecture underlying the system, and look for points of overlap or access where you might gather data.

Finally, try building or accessing a similar environment. If your teams were following standard software development life cycles, they probably needed testing and staging environments. If you already have those built, you can try simulating the same attempts at hitting endpoints to understand what a correct response should be. You can try shutting off different pieces of an API, for example, or upgrading an underlying operating system to see if a breaking change to container networking may have caused a container or application failure. If you don't have such a system, a skeleton system stood up for temporary troubleshooting can be just as valuable for quick diagnosis of a broken production system.

Hopefully, these basic ideas on how to gather data from your servers or services will give you just enough information to understand what went wrong and point you in the right direction to finish debugging your systems and get production back up and running. If none of this worked, have you tried turning it off and turning it back on again (after backing it up)?

Use Checklists to Manage Risk

Lisa Huynh

Lead Software Engineer at Storyblocks

Whenever there's change, there's risk. And when we're working in the cloud, we multiply that complexity as we add reliance on external resources that are changing underneath us. One tool that should be in your arsenal is a basic checklist, which will make running tasks simpler and more repeatable for you and the rest of your team.

The human mind can remember only a handful of items at a time. And when we do something like an infrastructure move, we have a million moving pieces to keep track of. Tasks run the gamut from getting teams' sign-off to ensuring that a flag for some minute but very important resource gets flipped.

And if something goes wrong, it's much less stressful if we already have a playbook prepared to handle failure scenarios instead of trying to figure out what to do on the fly. Skip the time spent on things like debating whether to roll back or hotfix by making the choice beforehand.

Enumerating the steps for debugging or maintaining the system is helpful in general. The checklist doubles as a form of living documentation for the expected behavior of the system, putting in writing what can often be tribal knowledge. Some of the annoying issues are those that get silently fixed, but then when the expert is away, you're stuck combing through logs or debugging from scratch to figure out what to do.

Imagine that an alert has come in that users are receiving a high number of HTTP server errors. The alert may link to a debugging checklist that looks something like this:

1. Check whether recent changes were made to the application and notify change owners.

2. Check load balancer metrics at <location> to verify that all servers are passing liveness checks and receiving traffic.
3. Check request metrics at <location> for changes in size or shape of traffic.
4. If errors do not appear to have been triggered by application changes, scale up the number of instances.

   If errors are alleviated, investigate autoscaling triggers.

   If errors do not occur on new instances, but continue to occur on old instances, cycle out the old instances.
5. Follow application debugging steps at <location>.

As you create these lists, keep a few things in mind:

Presentation matters.
: The longer a checklist is, the more likely it is that steps will be missed. If a list becomes long, we can break it into sublists. Sequential tasks should be listed in the order they are performed. For nonsequential tasks, critical items should be first, so they're less likely to be missed. Similar tasks should be grouped together in order to avoid context switching. In this case, we've broken out the application debugging list, and the steps have been ordered to eliminate the most areas at a time.

Be explicit.
: If you are writing a list to help your team respond to an issue, the steps are no good if only one person understands them. Avoid using jargon or abbreviations. Link to relevant resources as much as possible. Ideally, someone should be able to jump in and run through the actions with minimal hand-holding.

Automate as much as possible.
: When a task becomes routine, we can codify the work it is doing and automate the process. For example, if one of the steps is to cycle the machines, that's something we can probably instrument a tool to run.

No matter your team's size or budget, checklists are a valuable tool to hold in your pocket. Combat increasingly complicated systems by planning and documenting your process for change. Your team, and the future you, will thank you.

Everything Is a DNS Problem: How to (Im)prove

Michael Friedrich

Developer Evangelist at GitLab

"Everything is a DNS problem." This likely sounds familiar, and you've probably heard it from your team members during an incident. Customers say that your shop website is not working. You have verified it in your browser, and it seems fine.

DNS relies on a distributed environment, which makes it highly available and sometimes hard to debug. The problem is not necessarily bound to resolving a domain name. Instead, you need to get the whole picture with network routing and analyze the packet path with a client's point of view.

If the website is not reachable, is it a DNS problem? It could also be related to a specific location, where the client is trying to fetch the content behind a firewall or proxy. This needs more investigative work. Being the detective when your customers call can be a challenge.

Before you start analyzing the problem, stop for a moment and breathe. You cannot know everything. The documentation and Google are both your best friends. Forget what others say about "read the f***ing manual" (RTFM); the documentation should always be your first entry point. Maybe a friendly developer added a troubleshooting section, or you'll find a link to a page that explains the problem you're facing.

Accelerate your "Google-fu" by searching for the full error message and using multiple keywords for context-specific filtering. Find a partner on your team, and ask for help and guidance. Provide them with the complete context of your research. Take notes on the issue while the incident progresses. Whenever possible, share your screen in a call and do a pair debug session. Analyze the network routes together and learn more about DNS tracing.

Adopt a strategy to isolate the problem and cut it into pieces, narrowing the possible influencers. After a while you'll do this naturally, making you feel more confident when approaching a problem.

Do not depend on "fancy" tools; get comfortable in the shell with `find`, `grep`, `sed`, and `awk`. Practice searching the logs from the command line and learn vi(m), as it might be your only option for editing a file on a server or in the shell.

If you need to fix something in production, document the change and raise the issue with your team members. Adjust monitoring downtimes and alert thresholds during this time to avoid on-call chaos.

Keep practicing with different Linux distributions in VMs, containers, or CI/CD jobs. There might be a strategic decision requiring you to quickly adopt a new Linux distribution, or change your infrastructure-as-code patterns.

Make distributed monitoring and observability a priority whenever something gets deployed. Discover only metrics, traces, and states you need. Do not collect everything just because it is there. Document these strategies to educate your team.

Let's say the error in the shop website case turns out to be a load balancer where Kubernetes pods were not answering DNS queries, but only for a specific Internet Protocol (IP) source origin. Does it happen again after you adopt deployment and monitoring strategies?

Because DNS is distributed in your cloud native environment, you'll need good strategies to understand and resolve problems. Take incidents and failures as opportunities to learn. Develop strategies to battle-test your environment with chaos testing. Document everything in your handbook and create a postmortem analysis.

What's the Time?

Nikhil Nanivadekar

Director at BNY Mellon

Time is "the indefinite continued progress of existence and events that occur in an apparently irreversible succession from the past, through the present, into the future," according to Wikipedia. From a computing perspective, time can be a duration, or a date-time representation. Why is time an interesting problem in the field of computing?

Consider a day: each day has 24 hours, each hour has 60 minutes, each minute has 60 seconds, and each second has 1,000 milliseconds. Hence, questions like "How long did a process take?" or "How much time does it take to complete a task?" deal with duration. *Duration* can be computed by using this formula:

$$\text{Duration} = \text{End Time} - \text{Start Time}$$

Now let's look at time as a date-time representation. The representation consists of year, month, day, hour, minute, second. Using an ISO 8601 standard, time can be represented as a value like 2020-06-10T00:51:23Z, where Z denotes the time zone—in this case, Coordinated Universal Time (UTC).

Reconsider the preceding formula for duration; the seemingly straightforward formula becomes a little more complicated because of the time zones involved. For example, say a process is started by a user in India at 08:00 local time and completed by another user in New York at 08:00 local time. The preceding formula would indicate the process took 0 minutes. But that is not true; the duration of the task was either 9 hours 30 minutes or 10 hours 30 minutes, depending on the time of year the task was performed. That brings us to the next twist in the tale of time, called *daylight savings time* or *summer time*. So, the correct formula for duration is as follows:

$$\text{Duration} = (\text{End Time})_{@\text{TimezoneT}} - (\text{Start Time})_{@\text{TimezoneT}}$$

This fluid nature of time creates multiple challenges, especially when working with processes in the cloud. Imagine a process running in a datacenter in New York interacting with another process running in a datacenter in London. If both of these processes do the computation in their own time zone and share time-related information with each other, it can be a source of errors and bugs. Moreover, if the process in New York gets replaced by another process running in a datacenter in California, the time zone for the process also changes. The datacenter changes can lead to more possible problems when working with time. To circumvent these issues, it is a best practice to first convert all date-time information to UTC and then perform the computations. Another best practice is to share date-time information in UTC. Lastly, all date-time information should be accompanied by the time zone to reduce ambiguity.

Time can be extremely mysterious and can lead to many unexpected and unforeseen issues. The best way to work with time is to always consider time zones, and retain and share the time zone information during interactions. Time can be abstract and complicated, but it can be straightforward and simple too, so always remember to check the time with the time zone.

Monitor Your Model Dependencies!

Ori Cohen

Lead Data Scientist at New Relic

Algorithms rely on various packages, and each one of these packages will have multiple versions throughout its lifetime.[1] The following are several situations, out of many, where dependencies may break your model or deployment—and when that happens, you'll wish you had some sort of mechanism to monitor your dependencies, in addition to alerts that notify you when and where a problem has happened.

At times you will have *requirements.txt* discrepancies between environments. For example, your research environment is always up-to-date when it comes to scientific packages, such as pandas and scikit-learn, but your deployment has a lock on a specific version. Once new code or a serialized model is uploaded to staging or production, packages such as pickle or joblib will break when deserializing a model that was serialized using older package versions.

You may be using DeepMoji, a package that converts emojis to text in a preprocessing stage; for example, converting a 😃 to *happy*. Your NLP model relies on a certain mapping of emojis to text and was measured against a specific DeepMoji version. Once you update to a newer version, the mapping will change and directly affect your model, which in turn may lead to unexpected classifications, or simply influencing prediction probabilities.

Someone outside your team might tweak a package under their responsibility; this may happen up- or downstream, and it ultimately changes or breaks the expected functionality of your model. If it breaks, the solution is a matter of figuring out where the exception came from—and we all know that doing so is time-consuming, especially when the exception is external to your

1 A version of this article was originally published at Towards Data Science.

codebase. On the other hand, if it keeps on ticking, you won't be aware of the problem until a client complains.

To take another example, let's say a new deployment flow was recently introduced to your pipeline and for some reason, it breaks. The system automatically falls back on a previous flow that was deprecated months ago, but it's still pointing to old code, old dependencies, and old serialized models. Your clients will be served with predictions based on stale data, and information from new clients will be nonexistent in these old models. In other words, you will have a small catastrophe on your hands.

To maintain deterministic behavior, one solution is to record dependency versions in every environment and create alerts when a dependency mismatch occurs between environments. Another is to map dependency versions to your predictions, so you can figure out what went wrong, when it went wrong, and where it went wrong, and deal with it quickly and easily.

Unless you closely monitor your model's performance and dependencies and talk to your clients, you might miss these types of problems until something completely breaks, even if your unit tests pass with flying colors and your algorithm is technically functional.

In all of these cases, monitoring your dependencies in each environment will save you expensive work-hours by shortening time to detection as well as time to response and resolution. Most important, this approach will shorten the time during which your clients are negatively influenced by these incidents.

There's No Such Thing as a Development Environment

Peter McCool

DevOps Manager at CT4

…it's a candidate production system.

First up, a *candidate* production system isn't the same as a real live production system—not even close. It has, however, reached the stage where you can do yourself, and your customers, a big favor by starting to think about what life will look like when it becomes one. Systems can get to this stage awfully quickly, which can easily catch you by surprise.

There's a tendency, certainly on my part, to look at a development environment as nothing more than a piece of scaffolding: it's there to help me write some code; then it gets torn down, and everyone moves on. This is a common assumption, completely understandable and replete with really unappealing implications. I contend we are all much better off looking at these environments as candidate production environments.

For one thing, systems have a habit of escaping. You may write a system thinking you'll use it to do this one thing once. Then once turns into twice, and a year later, you've been using it to bill your real live customers for, like, a whole year—and it just sort of happened, both the system and the fact that it got so intimately involved with asking actual people to pay actual money. At least, that's what my uncle's dog's best friend's piano teacher tells me. Yeah. They also tell me they'd rather have thought about how it'd work as a production system, like, *at all* before it just sort of became one.

Another point is that *production system* really means "system that someone else, anyone else, can hold you accountable for," and this is a Rubicon that systems cross surprisingly early in their lives. OK, it may not matter to many people; their expectations may be low, and they may be understanding of lapses. But the point remains: this now matters to them. Build infrastructure is a good example of this sort of thing; it can matter to an awful lot of people

long before whatever is being built even looks like going live. Try telling a development team they can't deploy any code for a week and see how that goes over if you don't believe me.

There's more to it than that, though. If you're making your development systems as production-like as practicable (and you are, right? Right?), significant opportunities are available to build, test, and refine the production procedures as early as possible. Going live with something usually involves a week to a month of misery. You can choose to either spread it out over the preceding six months, whereupon it becomes less misery and more mild annoyance, or you can go with actual authentic misery when you go live and, worse still, straight after you go live. I've tried both; I much prefer any amount of mild annoyance to misery.

The obvious thing here is build and deployment automation. If you commit to a uniform process across all environments, you can get started on that stuff really early. That's not the end of the story, though. Logging and monitoring more generally also benefit from as much road testing as possible. These are all things that benefit from being built in, so that's another compelling advantage to thinking about them on day zero.

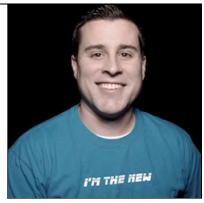
Potential cultural benefits exist too. When you start thinking about a system as a live operational system, you're probably tempted to talk to whoever it is who deals with that stuff, be it the SREs or the support team or whoever. Succumb to this temptation; that's my advice. Actually, my advice is twofold: arrange it so you're tempted as early as possible, and be sure to succumb right away.

This can all be summed up as *early engagement*. Engage early with the issues (the ones I mention don't constitute an exhaustive list, either), and with the people who might care about the system, either because they support it or because they use it.

Incident Analysis and Chaos Engineering: Complementary Practices

Ryan Frantz

Software and Operations Engineer



Learning from failure is extremely useful in a world driven by demands for high reliability.[1] Our approach to learning, and which failures we should focus on, may not be clear. Two common avenues to learning are incident analysis and chaos engineering.

Incident Analysis

Well-done incident analysis takes time. Constraining ourselves to a few hours or filling out a template to complete an investigation does not result in high-quality analysis. I've observed three outcomes associated with incident analyses: pushing paper, technical teaching, and surfacing surprises.

Pushing Paper

Regulatory requirements or fiduciary responsibility may require your organization to produce reports. In these cases, the value of analysis is largely superficial: it's part of a process to generate a paper trail. It's a bureaucratic, defensive procedure. Incident analyses are written to be filed, not read.

Technical Teaching

Imagine you operate an ecommerce site whose API fell over as requests stacked up. Your analysis of this incident is an opportunity to update documentation and procedures that can address technical gaps in people's understanding of the architecture. This is a way of building systems to be robust in the face of past known conditions.

1 A version of this article was originally published attention LFI.

The value of analysis seems tangible and immediate if we view failure as a chance to fill in some blanks in our processes, retrain ourselves, and possibly refine our practices.

Surfacing Surprise

A common theme across incidents is that something, somewhere, was surprising. Surprise is born from the events of an incident contrasting with our mental model of how the system operates. Focusing on surprise makes explicit the difference in mental models across a team of responders.

Beyond the specific details of an incident, other interesting surprises may be uncovered, including tension between business goals and procedure (i.e., work-as-imagined and work-as-done). Such deep insights about the nature of the work can allow organizations to make positive improvements, such as enhancing procedures to take advantage of efficient workarounds that have arisen to support underspecified tasks.

Chaos Engineering

Software is designed to express specific characteristics and behave in expected ways. Eventually, things will change enough that our initial designs may no longer serve us. *Chaos engineering* experiments help us evaluate our systems under realistic conditions so we can identify where entropy is slipping in and where our initial expectations are holding up.

Chaos engineering experiments have a similar value as incident analyses that provide technical teachings: the results are tangible and immediate. Chaos experiments are narrowly scoped, and the deliverable is clear.

Incident Analysis or Chaos Engineering

So should we spend time analyzing incidents or performing chaos experiments? Practicing one can inform the other. Incident analysis and chaos engineering are complementary, and when paired, may produce powerful results.

Recouping our Investments

The folks at Adaptive Capacity Labs have this to say about incidents:

> Incidents are unplanned investments.... Your challenge is to maximize the ROI.

Incidents inherently have value. Failures direct our attention to areas that warrant further inspection, including, possibly, via chaos experiments. Looking across a history of incidents, patterns may emerge that inform our designs. Through incident analysis, we can develop well-targeted chaos experiments that help us maximize the returns on our incidents-as-investments.

A Vision for the Future

In 1995, Boeing set out to test the wing deflection of its 777 airplane. The test subjected the wings to stresses well beyond what they were expected to encounter in operation—it was designed to find the wings' literal breaking point. The goal of purposely destroying a multimillion-dollar vehicle was driven by the aviation industry's history of analyzing and learning from incidents.

I have a vision for the future of software engineering: every organization will know the value of incident analysis, and that knowledge will be expressed in day-to-day work as it was for Boeing during this test 25 years ago.

Our job now, as software engineers, managers, or investigators, is to develop the skills necessary to analyze our incidents and draw out the deep insights they expose.

How Should I Organize My AWS Accounts?

Stephen Kuenzli

Founder of k9 Security

The most fundamental tools for organizing and protecting cloud resources are accounts: AWS accounts, GCP projects, and Azure subscriptions. Cloud accounts are architectural elements that create management, fault, and security boundaries. But many organizations do not use them properly, which puts the organization and its customers at risk.

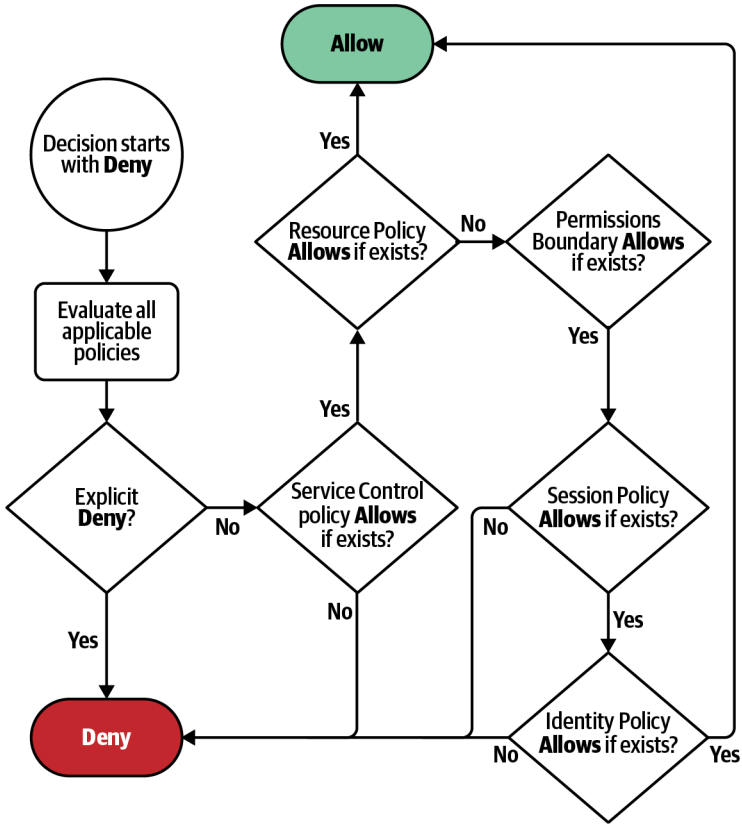
The rule is: create a cloud account for each major use case your organization operates in the cloud.

I'll show how to organize a large organization's cloud accounts to deliver changes and operate safely. Tailor this to fit your needs. Let's start with use cases shared across the organization, and then examine those for running end-user applications.

Every enterprise must support several use cases in their cloud deployment (the green in blue/green deployment). Provision the following accounts:

- The Security account contains the organization's Cloud API activity logs (CloudTrail) and resource configuration inventory (Config). Ingest these logs into log search tools in the Shared Services account.
- Operate monitoring, logging, DNS, directory, and security tools in a Shared Services account. Collect telemetry from the cloud provider, your infrastructure, and your applications running in other accounts. People with high privileges in other accounts may use this data and these services, but should not be able to modify operational telemetry.
- The Delivery account operates the powerful CI/CD systems that build applications and manage infrastructure. Operating CI/CD in a dedicated account simplifies securing that function.

- Create Runtime accounts for each business unit to develop, test, and operate its applications (the blue in blue/green deployment). Optionally, create accounts for the sandbox and disaster recovery.



Let's see how this structure influences autonomy, security, and cost.

Most organizations have multiple business units. Provisioning Runtime accounts for each business unit decouples decision making and access management between business units. This provides the freedom necessary for business units to get their jobs done with minimal coordination. Recognize that these choices will guide relationships between people and services within the enterprise going forward.

Architecture, team structure, deployment, and operational practices that do the work to deliver applications vary across business units. Recognizing and accepting differences helps business units coexist and adopt the cloud in harmony instead of battling over standards.

IAM users, roles, and policies are scoped to an account. Consequently, an engineer or application in one business unit can use resources without affecting another business unit. This limits risk of security compromises too. An attacker with a foothold in one business unit cannot automatically access another. Cross-account access can be enabled but this must be done explicitly.

Tracking and managing AWS operational costs at the business unit level will be very easy in both AWS and third-party cloud cost management tooling.

Most organizations have multiple phases of application delivery. Condense environments by purpose and deploy each environment into a separate account: dev, stage, and prod.

Application development teams can deploy changes and get feedback rapidly without fear of breaking downstream environments, particularly production.

Varying a person's permissions by delivery phase is straightforward when each phase has its own account. An IAM user or role in the dev account won't automatically get the same permissions in stage or prod. This simplifies giving the right level of access to data and operations at each phase of delivery. Deleting databases may be OK in dev, but almost never is in prod.

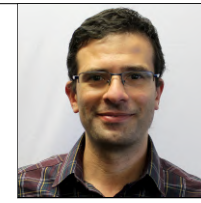
Partitioning accounts by delivery phase also demarcates audit boundaries and keeps non-prod out of scope. Partitioning by phase helps you understand how money is spent on each environment and set resource usage limits appropriately.

Organize cloud accounts to support the distinct use cases, structure, and delivery processes of your organization. Partition use cases by account to create safe boundaries for activities and data that enable your organization to move quickly and safely.

Resiliency and Scalability Are Key

Tidjani Belmansour

Cloud Solutions Architect at Cofomo



The rise of the cloud promises a virtually unlimited pool of resources. This opens up a whole range of new opportunities for everyone: the hobbyist, the freelancer, the startup, all the way to the biggest companies in the world.

Suddenly, the infrastructure running our applications can scale from one instance to thousands in a matter of minutes, if not seconds, in order to meet users' demand. Scaling should be designed both ways: *out* (when there's an increase in demand) and *in* (when demand decreases). This approach is also known as *horizontal scaling*.

Scaling out and *scaling in* refer to increasing and decreasing the number of instances in order to meet the demand in processing power so that users' requests are not only fulfilled (i.e., not rejected because of the servers falling under pressure), but fulfilled within a reasonable amount of time (we refer to this as *reducing the latency of our applications*).

Another approach to scaling is known as *vertical scaling*. With this approach, we increase (*scale up*) or decrease (*scale down*) the computing power of our instances (more CPU, more RAM, etc.) rather than their count.

Ideally, we should aim for horizontal rather than vertical scaling, for at least these two reasons:

- Vertical scaling doesn't increase the number of instances: thus, if we have only one instance of our service and that instance goes down, it may cause the failure of the whole system (if it's not resilient).
- Vertical scaling doesn't always guarantee that more requests will be handled: physical hardware limits increasing the capabilities of our servers.

Why is reducing this latency so important? Because it has been demonstrated that the longer your users wait for a response from the server, the more likely they are to simply abandon using your services and move on to using your competitors' services.

Resiliency and scalability are somehow related. But what does resiliency mean? *Resiliency* can be defined as the ability of a system to recover from difficulties. How is resiliency related to scalability? A system that is running under pressure could reject an incoming request even if it has already triggered an autoscaling request (which may take some time to be completed). If our system is not resilient, it will crash, and the user may lose their session, which is certainly frustrating. If the system is an ecommerce site, we are likely to have lost a sale! However, if that same system was resilient, the failed request would have been captured, and the system would have tried again a given number of times and ultimately succeeded in processing the request.

Back to the promise we've talked about. There's no magic; it can't be turned into an opportunity unless your applications can handle the scalability and resiliency of the cloud platform. This can be achieved only by designing applications with resiliency and scalability in mind.You can do that by using cloud patterns and tools/frameworks that support these concepts. Such tools and frameworks exist for almost every technology and programming language.

Infusing scalability and resiliency into on-premises applications has often been neglected for various reasons, the most common being that the maximum load of the system is known ahead of time and the system is designed according to it. However, the cloud has changed this, for the worse or the better: our systems may experience much more success than we've anticipated. If we designed our system so that it can handle such success, we may win big. Otherwise, our business will be negatively impacted.

So, next time you design a new application or rearchitect an existing one, consider using cloud patterns, and infuse resiliency and scalability at the core of the application. This will greatly benefit your applications, whether you run them on premises or in the cloud.

Monitor, You Will

Tidjani Belmansour

Cloud Solutions Architect at Cofomo

So, you have built a shiny new application that you plan to deploy to the cloud. You have applied all the best design patterns and practices to create a resilient and scalable architecture. You have tested your application using various methods and approaches in order to ensure that it meets users' demands and that it is bug-free—or, at least, free of "severity 1" types of bugs (you may have kept track of less-critical bugs in a "technical debt" registry of some sort). You probably also have scripted your infrastructure and created the required CI/CD pipelines. You've just deployed your application into the production environment, and you're ready to celebrate your success.

Well, not so fast. Haven't you forgotten something? What about monitoring?

What Is Monitoring and Why Should We Care?

No matter where you look up the word *monitoring*, you'll end up with a definition that is close to this one:

> Monitoring is the systematic and periodic process of collecting, analyzing, and using information to track the usage, quality, or progress of an asset toward reaching its objectives.

Monitoring requires data. It is the data that is gathered. It can be in multiple forms and come from various sources (activity logs, server logs, application logs, and so on). This is usually referred to as *telemetry data*.

This data includes, but is not limited to, information that answers questions such as: What action was performed? By whom? When? And what resources were affected by that action?

Is Monitoring Required Only for Cloud-Based Applications?

You've guessed it: the answer is no. Monitoring is important whether your application is deployed on premises or in the cloud, although organizations

may give special care to monitoring cloud-based applications as they are running outside their datacenters.

That being said, cloud providers offer you a wide range of tools, such as application performance management (APM) tools, that make putting together a monitoring solution and alerting mechanisms easier than doing it all by yourself.

What Should We Monitor?

Monitoring is a driving factor that will ensure that your application is successful in the long run. You'll rely on monitoring as an indicator to figure out whether your application is still meeting users' demands in terms of the following:

Functionality
: Are there features that are no longer used or that are confusing users?

User experience
: Is your application responding fast enough? Are users experiencing issues?

Usage patterns
: How are your customers using your application?

Security
: Is the application under attack? Was the application hacked? Was there any data exfiltration?

Billing
: Is your application costing you more than expected? Can you reduce your service plan?

Platform's health status
: Are there any service failures on the cloud platform?

And almost any other kinds of insights you may think of.

Monitoring and Dashboarding

Your cloud provider of choice is likely to provide you with a way to build dashboards in its console. This is a great way to set up a visualization of what's going on with your software system (application and infrastructure) by pinning metrics and telemetry data.

We Should Design Our Applications for Monitoring from the Start

To be effective, monitoring can't be considered just at the end of the application's design process. It has to be considered from the beginning of the design journey. You have to think about what telemetry data you want to monitor, at which stage and in what form? Where will you send that telemetry data? Will you need to query the telemetry data in real time? Do you need to be alerted if something unusual or potentially dangerous is detected? At what frequency? These are just some of the questions you'll need to answer.

By now, you probably have a better view into what monitoring can bring to you and your applications. So, the next time you work on a cloud-based application (or an on-premises one, for that matter), make sure you infuse monitoring into it.

Reliable Systems Don't Happen by Accident

Zach Thomas

Team Lead for Service Reliability at Genesys

While we're designing intricate systems, beginning with the happy path can be a helpful simplification—but it's a big mistake to design *only* for the happy path. While it's true for any computer program, our problems multiply when we interconnect things in the cloud.

Here's a partial list of things that go wrong all the time:

- Something you want to reach over the network is unreachable.
- Something you want to reach over the network is unusually slow.
- Demand for your service suddenly overwhelms its capacity.
- Users create data payloads orders of magnitude larger than you expected.
- Your API requests are being throttled by your platform.

Among other implications, the cloud era means that operational concerns have become development concerns. Guarding against the unhappy path will make the difference between a reliable system and a smoking wreck.

Any part of your system that is without limits is a part that can bring down your system. This applies to everything from inputs you accept to the amount of time you wait for a response from a downstream system. Enforce cardinalities. Do you expect your customers to create thousands of entries in your content management system? Then don't make it possible for them to create billions. Another place to enforce limits is at the front door to your service; even with automated horizontal scaling, you must place limits on the number of requests you will accept. When your service is running at peak capacity, it is far better to reject new work than to accept it and fall over.

The Architecture Diagram Is Also a Map of Failure Modes

When you look at an architecture diagram with your reliability engineering hat on, you'll begin to see that every box is a subsystem that can fail, and every line is a communication path that can flake out on you. Enumerating these failure modes in your design documentation is a good idea. For every dependency, ask the following questions:

- Do you have a good, brisk time-out?
- What is the retry policy?
- Is there a circuit breaker?
- Is there a reasonable fallback value we can use in case of failure?
- Can we defer the work and try again later?

Asynchronous Communication Is a Friend of Cloud Reliability

Since everything you want to communicate with on a network can fail, synchronous requests are the most brittle of all. Whenever you can tolerate a little more latency, put requests into a queue, so that the consumer of that queue can do the work when it's ready. In case of trouble, the consumer can handle the highest-priority messages first. In case of an outage, the requests can be deferred until the system is healthy again.

Exercise Adverse Conditions

You can make educated guesses about how your service will perform under duress, but it's far better to put it through its paces in a series of controlled experiments. For any alerts that are configured to page your team, try to create the conditions that will trigger the alert. Practice your disaster recovery plans in a controlled environment *before* you need them in production.

What Is Toil, and Why Are SREs Obsessed with It?

Zachary Nickens

Site Reliability Engineer at Woolpert

Site reliability engineers love to hate toil, but what *is* toil? Why are SREs obsessed with removing toil? Site reliability engineering is what happens when you treat operations like a software problem. How do you treat ops like a software problem?

SRE can feel opaque, but in practice, it is the essence of engineering: remove inefficiencies in one component so that other components may perform quantifiably better. Software engineers want their code to be simple, fast, and reliable: bug and cruft free. SREs want operations to be bug and cruft free! Cruft and bugs in ops and infrastructure can be described in one word: *toil*. Toil is the kind of work tied to running a production service that tends to be manual, repetitive, automatable, tactical, and devoid of enduring value, and that scales linearly as a service grows. Toil is any engineering effort devoid of meaningful value.

If a piece of software is going to be used, we should make commitments, at a minimum to ourselves, that it is reliable, secure, and observable—but there is *no such thing* as 100% reliable or 100% secure. When an issue occurs, software engineers need to be able to identify the issue, remediate or recover, and restore service. Slowing down to find all potential problems before release isn't the answer. If we slow releases, we sacrifice velocity, and the features we spent engineering effort on don't get released. Increased velocity is what we want. We want to ship new features quickly and release often. The answer lies in automation—in removing all the toil from the process of getting software deployed. We need automated testing in CI/CD pipelines, automated infrastructure provisioning and control via infrastructure as code, and automated monitoring and alerting for when bad things happen. We

need to remove as much manual, repetitive, low-return work as possible, so we can spend our effort engineering new features and new software.

Toil isn't only a problem when working on and shipping features. When things go wrong, toil gets in the way of remediation and recovery. Debugging a broken deployment script or manually managing environment drift takes us away from positive work and forces us to focus on negatives. If we automate as many negatives as possible out of our equation, we get to spend more of our time on the positives. Removing toil from the entire software life cycle makes the entire life cycle quantifiably more efficient and effective, more reliable and secure. Removing toil makes the development experience more enjoyable. It makes deployments more enjoyable. Removing toil makes error remediation and incident response faster. Removing toil from the life cycle makes engineers happier, and happy engineers create better software!

PART VI

Software Development

The Cloud Doesn't Care if It Works on Your Machine

Alessandro Diaferia

Senior Software Engineer at Utmost

If your code successfully compiled and all the tests passed on your machine, you're only a tiny bit closer to success. The increasing complexity of the production environments in the cloud is rendering our development machines an ever less accurate representation of the context our code is going to run in. For this reason, testing a local machine is not enough to build confidence in the code you write. We, as engineers, need to widen our mindset and go outside the comfort zone of our IDEs or our development machines by deploying the software we build. This is the only way we can understand the implications of running our code in a complex environment like the cloud.

As the markets become more competitive and the frequency of change of customer needs increases, companies need to stay agile at delivering value to their client base. Deploying software can't be an afterthought. It can't be a painful and error-prone activity that gets left as a last necessary evil, maybe delegated to an entirely different team. Running the software in a production-like environment should be incorporated into every development cycle. This is the only way issues can be surfaced and tackled before it's too late. People writing code should also deploy it, understand what the implications are when running it in the cloud, and understand the maintenance effort that their code requires. Every engineer working with systems in the cloud should become comfortable using tools for container orchestration, infrastructure as code, and deployment to test their work in environments that resemble production as much as possible. Deploying should be an inherent part of the daily work of building software for a software engineer working in the cloud.

In addition to continuously deploying, companies working at scale in the cloud will find that being able to *test in production* is a competitive advantage. The complexity of the traffic patterns going through high-scale production systems is increasingly harder to reproduce in a controlled testing

environment. Building and maintaining complex controlled test environments is not a fruitful investment. For this reason, organizations should rather invest in tools that help engineers to *experiment in production* and validate the underlying assumptions their code builds upon against real usage scenarios. Altering software behavior should be as easy as flicking a switch on the dashboard.

Luckily, we've never had so many tools available to help with all of this. Automation software delivery pipelines, container orchestration and security scanning, infrastructure as code, monitoring, alerting—all of this is available almost for free to anybody. Software engineers working in the cloud should flex their production operation muscles and embrace these tools as part of their daily work. There are no excuses for not doing this in an era in which the pace of delivery and speed of adaptability are key factors in the success or failure of a company.

It works on my machine! has never been as irrelevant a claim as it is nowadays.

KISS It

Chris Proto

Senior Cloud Engineer and Owner of DevOps Gorilla LLC

Most of us have probably heard the famous acronym *KISS*, or *keep it simple, stupid*. Whether you the realize it or not, you apply this fantastic guiding principle in your life.

When you go to the grocery store, do you exclusively buy exotic ingredients? Do you use only complex kitchen appliances at home? Unless you are like my late grandfather, whose kitchen looked like the inside of an "As Seen on TV" store, of course, the answer is no. What's much more likely is that your kitchen looks similar to mine and is organized (or disorganized) with simple, multitasking tools like knives, cutting boards, pots, and pans to prepare your repertoire of go-to meals (think Taco Night). Expensive ingredients and complex tools occasionally appear to help cook trickier meals, but what a pain it would be to cook complicated meals for every breakfast, lunch, and dinner.

It's such a natural idea, but we seem to have the opposite urge when it comes to our work in IT. Why do we find it so difficult to KISS in IT?

Although the concept of KISS is ancient, the specific phrase has its roots in engineering. The story goes that an aircraft engineer by the name of Kelly Johnson coined the term while working at Lockheed Martin. During this time, Johnson gathered a team of engineers tasked with designing jet aircraft. He proceeded to hand out a small set of basic mechanics tools and challenged them to design the aircraft in such a way that an average mechanic with this same set of tools could perform the repairs. Therefore, the engineers' designs had to consider both how the aircraft could break and how those breakages could be repaired "simple, stupid."

Cloud native systems are not jet aircraft, but they are complex, they're interconnected, and they will break. It's not a matter of *if* they will break, but *when*. As we design and architect systems, we become experts in those systems. We know the intricacies of the design, but we forget that our creations will continue on after us and need to be maintained by others who cannot be expected to have the same familiarity. As cloud engineers, we need to take

seriously our role in designing and building systems that are going to fail. We need to consider how to prepare for failure by creating systems that someone else can reasonably repair when they break, using only a simple set of tools and a decent understanding of the system. Only then can we ensure that Johnson's "simple, stupid" approach lives on in cloud engineering culture.

Here are some ideas that will help you apply KISS in your cloud engineering practices:

- Avoid premature optimizations.
- When you need to make complex optimizations, provide detailed documentation.
- Start small and use minimum viable products (MVPs) to help guide design decisions.
- Read the documentation to understand the cloud APIs you consume. Pay close attention to rate limits and error codes.
- Focus on learning best practices and avoid needlessly complex and confusing systems.
- Remember that your idea of what's simple is different from that of your operators.
- Use standardized naming conventions that provide context.
- Make time to delete unused cloud resources so they can't distract from what's important.
- Understand who's operating your system and their capabilities.
- Find your system's failure scenarios and provide runbooks to resolve them.
- When in doubt, take Albert Einstein's advice: "Make everything as simple as possible, but not simpler."

Good luck, happy clouding, and don't forget to KISS it!

Maintaining Service Levels with Feature Flags

Dawn Parzych

Developer Advocate at LaunchDarkly

Cloud engineers are responsible for designing, monitoring, and deploying applications to the cloud. These applications must be scalable, reliable, available, and fault-tolerant. That's no small feat. The need to continuously integrate and deploy new features may be at odds with maintaining acceptable service levels.

Customer demand for new features and capabilities is driving companies to push features out faster, but those same customers also expect available and reliable application performance. How do you balance these conflicting priorities? Implementing a CI/CD pipeline with a battery of automated tests is the first step. But as applications grow, so will the differences between staging and production—impacting your ability to find all bugs before your customers do.

Building a CI/CD pipeline is not enough because staging is not production. You need to put safeguards in place to safely deploy, test, and release features in production without negatively impacting customers and the bottom line.

Pushing out new features can result in service outages, no matter how much testing you perform. You need to be able to deploy code without releasing it to all users via feature flags.

Decoupling feature releases and code deploys with feature flags makes it possible for you to do the following:

- Progressively roll out a new feature via:
  - *Ring deployments or canary releases*, whereby different groups of users gradually receive the feature for testing and to manage the risk. For example, first the development team has access, then internal users, and finally all users.

— *Betas,* whereby preselected users receive access to new features to provide feedback on existing functionality, identify bugs, and suggest new functionality.

- Utilize circuit breakers and kill switches to turn off a poorly performing feature without rolling back to a previous release.
- Test in production to validate the interoperability of services.
- Synchronize the rollout of a feature that requires changes in many components or microservices.

After deploying code, you need monitoring and observability tools to collect diagnostic data and inform you of whether the application is performing as expected. If something goes wrong, an alert needs to be triggered that kicks off an incident response process to minimize the impact on users.

Feature flags or toggles are a critical piece to this process, enabling teams to move faster, reduce risk, and maintain control. A feature flag is an if-then statement.

If a user meets a set of criteria, that user gets access to the feature. For example, if you are running a beta for a new chat widget in your application, only members of the beta group should have access to the widget. A feature flag can control this behavior:

```
if (profile["beta"] == true) {
  displayWidget();
}
```

When getting started with feature flags, here are some tips to consider:

Start small.
: Choose one feature, a group of features, or a team to start. You can do a lot with feature flags. You will overwhelm yourself and others if you try to do everything at once. Beyond release management and operational efficiency, you can use feature flags for experimentation and entitlements. Once you have a robust feature flagging foundation, you can expand into those use cases.

Consider feature flags at the design stage.
: The right time to think about whether to implement a feature flag is when you first start designing and planning the feature. Design considerations should include naming conventions, who can toggle the flag, the purpose of the flag, and whether the flag is short-term or permanent.

Use feature flags with other tooling.

Think about how to use feature flags with existing tools in terms of creation, deletion, and toggling of flags. Put practices in place to toggle features from monitoring and incident management tools. When a feature is wrapped in a flag, you can release it to a small group of internal users initially. If monitoring and observability tools identify a problem, the feature can be toggled off manually or programmatically via the incident response process. Avoid technical debt by writing a pull request to remove a short-term flag at the same time the pull request to add the flag is created.

Feature flags are an essential part of the CI/CD pipeline for cloud engineers. They help you release features quickly without worrying about incidents negatively impacting service levels.

Working Upstream

Eric Sorenson

Technical Product Manager

Along with the rise of cloud computing has come the rise of foundation-managed open source. A huge number of valuable cloud-related projects exist, but using upstream open source software (OSS) is not without its problems. The old saying "There's no such thing as a free lunch" goes double for "free" software. Bugs inevitably bite, features don't work as advertised, and sometimes deeper design problems prevent a clean integration. These guidelines will help maximize the benefits and minimize the cost.

Let's assume you're working on a new initiative and want to incorporate an open source project as a significant part of the product. You hope this will let you speed up your time to market and focus in-house development on the features that add real value, but you're not sure how to get started.

Survey the Landscape

Depending on what you're working on, more than one project may be suitable to use. Start out by listing your top three to five requirements and scoring the projects with a low/medium/high ranking. Then add nonfunctional criteria:

How active is the project?
: Having hundreds of open issues isn't necessarily bad—just look at Kubernetes!—but if the repository has gone months (or years) without a release, that could be a warning sign.

How amenable is the project to external contributors?
: Look at the commit history to get an idea of how many people and organizations have contributed code.

What's the governance model?
: Some nominally open source projects have, in practice, a restrictive license, an onerous contribution model, or a decision-making process that favors a single vendor.

A single low score isn't necessarily disqualifying, but the aggregate should give you an idea of how suitable each project will be. Plus, your scoring provides a great lead-in for the next step.

Get Internal Approval

Hopefully, your company is amenable to the idea of open source software. If not, it might be time to look for another job. (Just kidding…maybe.) In any case, it's a good idea to present your ranked choices to your manager and team. The reality is that working effectively with open source *does* have a cost, and it's important to account for it early so you don't get the rug yanked out partway through.

Join the Community

A big part of that cost is the effort it takes to become a good community member. This is often called *chopping wood and carrying water*: unglamorous work to help the project and community. All projects need people who improve documentation, respond to new users on Slack, and maintain release automation. This has several benefits:

- You increase your familiarity with the codebase, which can help when you need to modify it.
- You gain reputational currency, making it more likely the community will help you.
- You improve the viability of the project overall, effectively investing in its future.

Design First, Then Code

As your usage of the project deepens, you may need a substantive new feature. There can be a strong temptation to keep your implementation local. While this approach may be quicker, resist the temptation and do the work upstream. Local changes mean you have to carry the patch to each new version, which becomes expensive and brittle over time.

To work upstream, instead of coding up the most convenient thing, start a design discussion with the project maintainers *before you write code*. Many projects have a process for this, like the Kubernetes Enhancement Proposals, which are required for complex changes. If your project isn't that formal, you can still use the template as a writing prompt.

This approach is counter to many developers' instincts, but it produces better outcomes. Because the discussion happens in the open, the design can take into account different perspectives, past experiences, and broader requirements.

Happy Upstreaming!

Open source can work like a mechanical flywheel for your development velocity: it takes an initial investment in energy to get the wheel spinning, but other people are helping with their own contributions too. Once it's activated, the momentum of the project can carry you to places that would be difficult to go alone.

Do More with Less

Ivan Krnić

Head of Software Development at CROZ

Cloud infrastructure introduces many benefits, such as elastic scaling, immutable deployments, and a pay-as-you-go pricing model. Pay-as-you-go is a huge advantage since we're paying only for what we're actually using, but also a liability if our applications are not as efficient as possible. It's not that we didn't want our applications to be efficient before the cloud—it's just that the pay-as-you-go model has made this even more important, since cloud usage is billed by actual resource usage.

The *traditional client/server programming model* relies on a thread pool. Whenever a client makes a request to the server, one thread from the server's thread pool is dedicated to processing that request. If that request processing includes a blocking call to an external resource, the whole thread is idling until that call is finished. Because one thread is dedicated to each client request, if there are more concurrent client requests than threads in the pool, excess client requests will be dropped. To increase the application capacity, we need to either increase the size of the thread pool or scale the application. In the former case, we're increasing memory usage. In the latter, we're also increasing the CPU usage. Both actions seriously impact cloud usage and are reflected in the pay-as-you-go pricing.

A better way of designing our applications that enables us to achieve better performance with fewer resources is called the *reactive programming model*. In this model, only one thread (event loop) listens to client requests. As soon as a request comes, it is dispatched to a specific event handler for processing in that same thread. The thread is never idling because all operations are implemented as nonblocking. If a blocking operation needs to be performed, the operation is called and the thread is immediately released for other client requests. When the blocking operation is finished, it throws an event signaling that a particular client request is ready to continue processing.

Although only one thread is needed for a reactive programming model, typical configurations use as many threads as there are available processor cores. That way, every thread executes on its dedicated core and there are no context switching penalties.

While the traditional programming model handles excessive client requests by simply dropping them, the reactive programming model supports a *backpressure* mechanism that controls producers and keeps them from overwhelming the consumer with too many requests. Backpressure is a far more appropriate method since no requests are dropped and the flow of value through the system is adjusted for optimal end-to-end flow instead of for maximizing local optima of each component in the processing flow. Backpressure aligns nicely with the theory of constraints. Once the optimal end-to-end flow is achieved, we can further identify system constraints (processing bottlenecks) and decide how to exploit them and possibly elevate them to deliver an even more efficient system.

The key advantage of the reactive programming model is *high efficiency*. With only a couple of threads, we can do as much as we could with a thread pool of 200 in the traditional programming model. Fewer threads mean lower memory consumption as well as more efficient and cheaper running in the cloud. Under light load, there will probably be no difference compared to the traditional model. But under heavy load, the differences are staggering: sometimes as much as 3× more efficient while using 50× fewer threads!

Everything Is Just Ones and Zeros

Lukas Ruebbelke

VP of Developer Growth at BrieBug

Programming is inherently complex because we are trying to capture and express a reality that is, by nature, very complex. To further complicate the issue, our exposure to the full nature of the complexities of our domain is constrained to a particular and limited context, which is heavily influenced by where we fall in the organizational chart. For instance, frontend developers are experts at building web applications, but rarely understand the infrastructure required to put those applications into the cloud. The immediate symptom of this condition is that we develop a myopic view of our world and establish causal relationships where none exist. The longer-term and ultimately more costly outcome is that we end up doing the same things over and over at every abstraction layer and calling them by different names.

As engineers, it is easy to get fixated on the ergonomics of a particular framework or technology and to project our frustrations onto the perceived deficiencies of whatever we are working with at the time. Though a reasonable reaction, it usually results in a causal association where there is a correlation at best. The most common example I see occurs when someone tries to explain why they do not have adequate test coverage in their project. The response usually goes something like, "Well, we would write more tests, but it's too hard because Some_Testing_Framework is impossible to get working!" I am almost always willing to bet money without seeing any project code that testing is so hard not because of bad technology, but because of bad code. *It is hard to write good tests for bad code!*

Once we step back from our current situation and examine the multiple dimensions that it encompasses, from the technical stack to the organizational hierarchy to the business domain, we can start to realize that common themes weave each layer together. As a software engineer, how do you know your code works? Your tests pass. As an engineering team, how do you know that you are building the correct thing for your stakeholders? By using Agile

software development. As a company, how do you know that you are bringing the right product to the market? Lean product development. At the highest level, how do we know anything? Feedback loops. *Unit tests, Agile software development, and Lean product development are all forms of feedback loops.*

Hardware has seen an unbelievable trajectory in terms of raw power, but at the end of the day, it is still a bunch of tiny little switches flipping on and off, making ones and zeros. The only difference is that the composition of these switches has become a little more sophisticated. Modern software has exploded in complexity as technology advances and markets evolve. Yet, at the end of the day, we are still using the same basic patterns over and over.

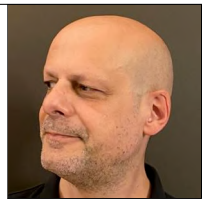
I believe we can summarize everything we do as programmers as four basic things: nouns, verbs, conditions, and iterations. *Nouns* are data structures that model our domain, while *verbs* are the methods that allow us to perform units of work. *Conditions* allow us to decide what work to do, and *iterations* allow us to do a unit of work more than once. That is the extent of what I do as a programmer. I can reduce every enterprise-level application that I have ever put into production down to these four things.

So what does this mean? If developers, especially aspiring ones, were to worry less about the hottest new framework on the block and instead focus on understanding where the "ones" and "zeros" are, they would be exponentially more adaptable and effective. Frameworks come and go. Languages come and go. Platforms come and go. *But first principles and patterns will always exist, because they are the ones and zeros.* When you start to look at everything in the context of the bigger picture, you realize everything is just ones and zeros. We just have different names for them.

Be Prepared to Repeat

Ricardo Miranda

Gig Data Engineer at Closer



My first car had a remote with a single button to lock/unlock the doors. The car behaved as a finite state machine: if the doors were locked, pressing the remote's button would unlock them; the next time, the doors would lock again. This was really annoying. How I wished the remote had distinct lock and unlock buttons!

According to Wikipedia, "*Idempotence* is the property of certain operations in mathematics and computer science whereby they can be applied multiple times without changing the result beyond the initial application."

In the cloud, systems tend to be decoupled, with components communicating among themselves via asynchronous message passing. Eventually, duplicate messages appear, either intentionally or unintentionally. By preparing for idempotency we can avoid costly refactoring.

Strategies to Cope with Duplicate Messages

Strategies to deal with duplicates vary in several dimensions. The first thing to consider is how expensive it is to compute more than once. Making services aware of which messages were already processed creates an overhead that should never be underestimated.

Often it is easier to just repeat operations. This is a big leap from the monolith (a single large application)—a leap fundamental to becoming a cloud engineer.

Stateless Consumers

Stateless consumers are servers that do not keep track of processed messages; every message is processed independently from previous and future messages. Every consumed message is processed, even if it is a duplicate. Algorithms and storage must be prepared for this. For example, imagine a service that processes data and stores results in a database. It may be necessary to

implement a cleanup stage before processing to avoid writing duplicates. Storage systems that do not allow updates may need custom solutions.

At times, duplicated results may not be a big issue. For instance, counting the number of hits on a web page may not require an exact number.

Whenever processing a message is irrelevant after a time frame, this information should be included in the message itself (preferably in the header). Most messaging systems have the concept of headers (or attributes), where metadata is attached to the message payload. Consumers can discard old messages.

Keeping State

Stateful servers are servers that keep track of processed messages. These messages should be identified with some metadata, not with the payload itself. This is very important to distinguish between repeated messages and messages with equal payloads. When creating a message, attach an identifier attribute that makes duplicates explicit.

As in stateless servers, if there is a time window after which processing a message is no longer required, create a time-to-live header so that old messages may be forgotten, reducing the effort of keeping state and checking whether a message was processed.

If the state is stored in an ephemeral structure, the same rules applied to stateless servers apply to stateful servers. If the state is lost or incomplete, the service may not be aware it is receiving a duplicate.

When a service with several instances processes messages in parallel, it may be necessary to share state among them, usually with the help of a database. The choice of the state persistence stack depends on the reliability/performance axis that is part of the decision making.

Conclusions

Cloud native computing is intrinsically distributed in nature. In distributed decoupled systems, it is extremely hard to be sure that messages are received exactly once and that retries are never necessary. It is simpler to assume, right from the project's inception, that repeated messages will eventually appear. Every step must include the assumption that a message may be a duplicate, and a strategy to deal with that should be in place.

Your Greatest Products Are Not the Applications and Services You Produce

Ryan Bell

Director of Creative Energy at Vim Labs

This may come as a shock if you've been in the industry for a long time, having learned to keep your head down and assume the role of a productive software developer, but you are not solely in the business of delivering applications and services. Software developers indeed develop software, but I believe that it is as imprecise a description of the nature of your true role as it would be for a musician to describe their calling as delivering etched vinyl and audio encodings to record labels. Applications and services are your media. They're a packaging layer that encases a greater offering you can extend out to the world. You are in the business of delivering uninterrupted Magic Moments, one after the next.

That first time you played *Super Mario Bros.* with your friends around a Nintendo Entertainment System and stumbled across a Fire Flower power-up—that was a Magic Moment. Soon after acquiring this new superpower, you were no longer pushing buttons on a square controller; you were joyously tossing fireballs across the screen like a boss, experiencing this magic uninterrupted. Now, remember that before that moment, your character was just an ordinary plumber navigating drainage pipes in an 8-bit world. There is a tremendous difference in mindset between solving the problem of efficiently animating pixels on a screen or moving sales units, versus the task of translating a brief feeling of infinite possibility through your software to the next generation of artists and problem solvers.

The greatest product you can deliver to your customers is the superpower. The power-up. The Magic Moment! These are found inside your applications and services, at the intersection between functionality, aesthetics, and

surprise and delight. It's when an end user suddenly encounters an unexpected flash "wow-that-is-really-f***ing-cool" moment, leaving a lasting impression, coloring the way they express themselves as they share this gift with their friends, family, and coworkers.

Should you choose to accept it, your mission is to design these moments, construct bridges to connect them, and smooth out the edges interrupting their flow. While this undertaking may entail code refactoring, microservice architecting, project management oversight, user interface/user experience (UI/UX) research, DevOps monitoring, unit testing, learning new languages, installing new frameworks, and navigating the surrounding territory of the software development landscape, these are not the goalposts. Instead, they line the field leading toward the ultimate magic destination.

You can envision a 2D chart being plotted of Magic Moments: Minor Annoyances / time, as end users experience your products over the lifetime of their service from one feature to the next. The equation for the best possible product that you can deliver can then be expressed as P = (MM / MA) / t × L. Rather than focusing on the how or the why, I propose keeping your eye on the *wow*.

From this new vantage point, what changes can you make to produce a shift from mere problem solving to superpower bestowing?

Avoid Big Rewrites

Simon Aronsson

Developer Advocate at Load Impact

We've all been there in some form at one point or another. Our product owner has asked us how long it would take to enable our software to run serverless.

For the last couple of hours, you've been browsing the code, trying to make sense of it. But it's just too old, too much of a patchwork. Layer after layer of added abstractions and cross-dependencies. Names of past engineers you've never heard of. It almost feels like you're about to break the F12 key on your keyboard just repeatedly calling Go to Definition.

At last, tired and full of frustration, you give up. You go to your product owner and say that the shape of the code is just too bad. That it would require a total rewrite. Two days later, the product owner gets back to you and gives you the go-ahead. Full of excitement, you assemble the team to tell them the great news…

I've been in this situation enough times to tell you that your chances of taking on a task like this and succeeding are slim to none. Every time I've been part of a project like this, we've always come to regret it. Common outputs have included (although not been limited to) the following:

- Not making deadlines
- Going over budget
- Introducing obscure bugs
- Burning out team members
- Losing stakeholder confidence

If the codebase is too complex or complicated to understand and refactor, it's *definitely* too complex or complicated to rewrite. So what to do?

Step 1: Be Realistic

It's almost always possible to migrate your workload to a virtual machine in the cloud as is. While this might not be exactly what you were hoping for, it will still allow you to leave the confinement of your local datacenter and leverage the benefits of not having to manage your hardware, or even OS, yourself.

Step 2: Utilize the Strangler Pattern

The *strangler pattern* is used to incrementally modify an existing system by extracting parts of it gradually. Say, for example, that your application is used for order management and consists of about 40 main workflows.

Instead of rewriting all 40 workflows at the same time, pick one with a limited scope and extract that into its own serverless function or microservice. Once this is done, redirect all printing requests to that function or microservice and remove the now redundant code from your monolithic codebase.

Step 3: Repeat

Rinse and repeat until the monolith has completely vanished, or you feel you're done extracting all the high-value, change-prone parts of your system. In addition to being a lot less risky, this will also allow you to abandon your migration efforts at any point with the system intact, still delivering as much business value as ever.

Lean QA: The QA Evolving in the DevOps World

Theresa Neate

QA Practice Lead at Slalom Build Australia

When you see DevOps being practiced, but the QAs (or testers) still relegated to the "checking things" corner and not explicitly and proactively involved in providing input into both applications and infrastructure code, ask yourself: are we perhaps doing the very opposite of what DevOps was meant to be?

Beware the Cargo Cult

Most of us know the tale of how DevOps was conceived, and I trust most of us read *The Phoenix Project* by Gene Kim et al. (IT Revolution Press, 2013) as one of our first DevOps books. Therein we learned that DevOps is meant to be a collaborative and efficient partnership between all disciplines to achieve a common goal, which ultimately means having working software in a production system, delivering value to customers.

When DevOps excludes other team members, like security or QA, we have missed the point. DevOps is meant to be development *teams* working with operations *teams*, not just individuals in developer and operations roles working together. The QA role is considered to be a member of the development team. The QA is therefore intrinsically a member of DevOps.

If QAs have no input into operations or continuous delivery conversations because they are "not developers," but we think we are doing DevOps, we are, in fact, practicing a *cargo cult*—an imitation and approximation of the real thing done without a real grasp of why we're doing it. If we are going to do DevOps, it behooves us to know *why* we're doing it, and not only *how*.

Waste

Agile software development and DevOps fundamentally assist with reducing waste. (I generally prefer not to use these capitalized words as they have become overloaded, so I use them here cautiously.)

Project churn, conflict, friction, slow responses, unreliable and unstable systems, needless expenses, lengthy handovers, defects, time delays, failed projects, manual overheads, double-handling, and many similar issues are all waste.

Every time a feature is double-handled (such as the churn when a bug is discovered later), we have incurred waste. Feedback should be sought earlier rather than later so that one may course correct or remedy issues as early as possible and prevent waste.

QA Is Feedback

QAs do not provide assurance; they help provide and define analysis and feedback. Monitoring is a form of feedback on the system's behavior. Just as monitoring is feedback, so is testing. Whether the testing is manual exploratory testing or automated checks, it is feedback.

Testing tells you about components and relationships; monitoring tells you about the system. To minimize and reduce waste, you want to receive that feedback as early as possible, and you want that feedback to occur at meaningful levels and depths in your system.

Early Feedback

QA (testing and monitoring) should be done early and continuously. Receiving early feedback enables you to act on that information and improve or course correct as early as possible.

Your QAs can and should be involved in these conversations. If they cannot be, then as their manager or peer, the onus is on you to empower them to become involved.

The days of QAs telling you whether you're OK to "go live" are numbered (image credit: Milly Rowett). The whole team should know that, based on the continuous feedback they have been receiving.

Lean QA

The reduction of waste (rework, defects, friction) by holistically testing the whole system, early and continuously (and into production)—including measuring what really matters and incrementally improving thereupon, and using your humans intelligently in doing so—is what I call *Lean QA*. And I consider it *essential* to DevOps.

Source Code Management for Software Delivery

Tiffany Jachja

Tech Evangelist at Harness

Source code management (SCM), also known as *version control*, allows engineers to manage their software code. SCM provides benefits for developers as they work on different parts of a codebase, collaborate, and deliver new software releases. Done right, SCM enables development teams to build applications while avoiding irreversible or breaking code changes.

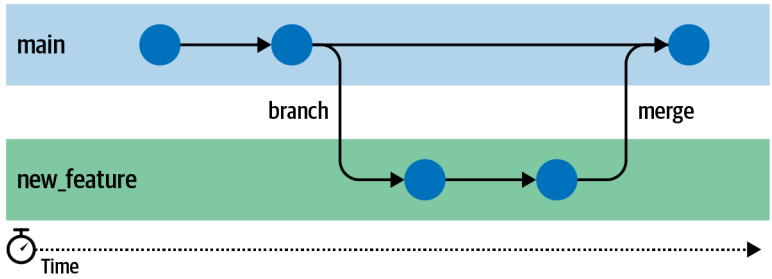
Understanding Version Control

SCM tools provide version control for reverting, tracking, and correcting software code. Think of version control as a timeline for your code revisions. A branch represents this timeline. In the most basic of version control workflows, you have the main branch. This main branch is often called a *trunk*.

You can progress within the timeline by making code *commits*. Code commits represent points within the timeline. Each point contains a copy of the current source code. A code *push* uploads these changes to a repository. The repository, or *repo*, holds your version-controlled code as a project.

A key concept is to introduce branches that spawn from points in time within the main branch. It is common to have a branch that contains feature development work.

For example, imagine a developer is working on a development feature for version 1.0 of their product. They want to develop and make changes to the codebase without affecting the main branch. Other developers depend on the main branch as a stable copy of their codebase. The developer decides to commit and push their changes to a different branch. This branch came from a copy of the code from the main branch. Once the feature development is done, they can merge their branch changes back into the main branch.



A best practice for using SCM is always to maintain the trunk with clean working versions of your code. The default branch for a repository is the main branch. Anyone who *clones* the repository code should not need to check out different branches to build the application on their machine.

It is also common practice to tag versions of code within the main branch. These tags indicate specific releases of code; for example, a stable 1.0 release, or a beta release. Anyone with access to your repository can correlate a tag with a version of the codebase.

SCM keeps track of who made changes, where, and when within the codebase; it also allows engineers to compare code between commits or branches.

What Is Git?

Git is a popular SCM tool. Strategies for using Git, such as git-flow, emerged as version control and became adopted as software practice. However, Git strategies are not one size fits all.

It can be challenging to maintain and adapt certain structures and ways of working. For example, git-flow may not be a good fit for high-performing teams whose code needs to get out quickly and often. Because the *pull request* model enforces an extra layer of approval to merge code into the main branch, this adds an extra dependency to get to a software version release. As seen with open source, however, git-flow is a great branching strategy to manage larger projects with many contributors.

But it's not only about branching. SCM does not negate "best practices" for software development. Here are some considerations when working with SCM:

Determine how your use of SCM lines up with your work and team.
: How important is it to iterate quickly? Is this greenfield development? Is there a high percentage of change failure?

Have SCM rules.
: Set practices for the team on how and when to commit. Show an example of a commit message. Establish how you want the team to style commit messages to be informative and consistent.

Determine what gets stored in the repository.
: Where should you keep workspace configuration or infrastructure as code files? Find a structure that works for your ecosystem.

Consider integrating your SCM tool with your software delivery process.
: A code commit could be a great way to trigger your software delivery process or CI/CD pipeline, which can also tag release candidates for consistency.

A major component of delivering better is working with others in a team. SCM enables effective code revisions and versioning.

PART VII

Cloud Economics and Measuring Spend

FinOps: How Cloud Finance Management Can Save Your Cloud Program from Extinction

Deepak Ramchandani Vensi

Transformation Director at Contino

Time and again we see organizations run into some pretty big cloud finance headaches:[1]

- Cloud spending is much higher than expected.
- The cost benefit can't be seen from the datacenter lift-and-shift.
- Too much is spent on licenses and services that the organization doesn't have control over.

But the problem is not the cloud; *it's the cloud consumption model!* The organization's financial approach to consuming the cloud is inherently flawed.

The cloud has fundamentally shifted the way organizations purchase technology. But while organizations dedicate a lot of time and attention to transforming familiar disciplines such as engineering, security, governance, and operations to operate in a cloud-first world, their finance and procurement functions are still geared for consumption of traditional on-premises IT.

This causes a host of issues. A few common ones include the following:

- Inability to manage the shift from capital expenditures (CapEx) and operating expenditures (OpEx)
- Inability to easily forecast tech spending

1 A version of this article was originally published at Contino.

- Inability to control the self-service cloud consumption model

How can finance and procurement adapt to the new world of the cloud? Enter FinOps.

What Is FinOps?

FinOps is an approach to managing and operating cloud spending by breaking down the silos in engineering, finance, and procurement. FinOps is meant to drive a shift in culture in cloud financial management, akin to the way DevOps and SRE have driven a cultural change in engineering.

It aims to bring together all the key functions involved in planning, procuring, consuming, managing, and governing cloud services in order to make the right decisions and trade-offs when consuming the cloud without compromising on the consumer experience and value that the cloud was meant to offer in the first place.

FinOps prevents spurious cloud consumption patterns and optimizes cloud consumption. Ultimately, it could save a cloud program that otherwise would have been bogged down by spiraling costs.

Here are some core FinOps principles:

- Make finance and procurement part of the planning process with engineering teams, not the gatekeepers of cloud spending.
- Provide guardrails for shared financial accountability, which is federated out to product teams.
- Design and architect with finance in mind.
- Use financial tracing to align cloud spending to product and customer metrics.
- Provide real-time visibility of cloud spending for consuming teams.

These principles help you to optimize your cloud spending, empower your product teams, and make good financial decisions based on clear metrics.

How Do You Get Started with FinOps?

A fundamental mistake most organizations make is to consider cloud financial management and governance either as an afterthought or as a cost-saving activity.

However, at Contino, we take an engineering-first, data-led approach to FinOps. To build a successful FinOps function, we recommend starting off with the following building blocks:

Use a cloud cost-control or FinOps policy.
: This provides consuming teams with a control set to comply with in order to achieve the financially controlled consumption of cloud resources within the organization.

Have guardrails for the cost-control policy.
: Implement a key set of controls as code-based guardrails within the foundational cloud platform.

Establish a cloud benefits framework.
: Establish a data-driven framework that helps articulate the benefits and ROI that result from your cloud program.

Identify your cost drivers and metrics.
: Identify and list the key cost drivers for application/product teams so these can be traced back to business benefits and outcomes.

Offer visibility for everyone.
: Provide real-time visibility on cloud spending for the consuming teams, as it helps them make better decisions and understand the financial implications of their cloud usage.

Establish a financial trace to your customer experience.
: Just as engineering teams use application tracing to profile, monitor, diagnose, and pinpoint any application failures, FinOps teams need to be able to trace cloud spending to benefits and cost drivers.

Summary

Bringing all of these things together can help your business make data-driven decisions that take into account cloud financial data, business metrics, and organizational and consumer insights—and bring your finance and procurement teams with you into the world of cloud!

How Economies of Scale Work in the Cloud

Jon Moore

Chief Software Architect

Are you familiar with the economic theory of *experience curves* (also known as *learning curves*)?[1] For cloud computing, this theory explains not only why it makes sense to outsource new datacenter costs to public cloud providers, but also why it may make sense for you to stop operating a datacenter at all.

Experience curves were formalized by the Boston Consulting Group (BCG) and describe how production costs tend to fall in a predictable fashion as the number of units produced increases. Namely, the more you produce, the better/quicker/cheaper you get at it. This is the essence of *economy of scale*. These curves are usually formalized as a percentage cost: for example, a 75% experience curve means that with each doubling of production, the marginal cost of producing the last unit drops by 25%. So, for example, one unit might cost $100, but the second costs only $75. The fourth costs $56, the eighth $42, etc. Experience curves show a diminishing rate of return.

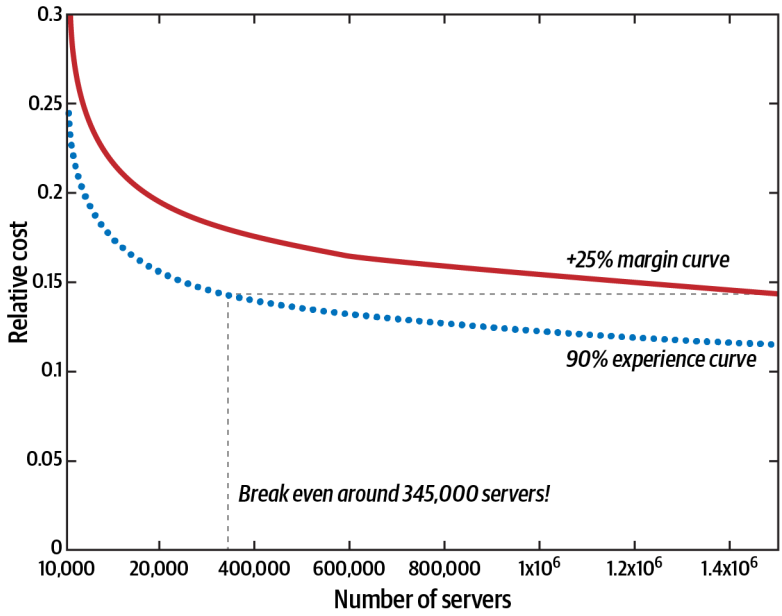
In the cloud computing case, we want to know the marginal cost of deploying and maintaining servers. While we don't know the actual learning rate, typical experience curves fall in the 75%–90% range. So let's assume datacenter server deployment follows a curve in that range as well.

Suppose you can rent a Linux virtual machine from your favorite public cloud provider for 10 cents per hour. We can assume, since that public cloud provider is a for-profit enterprise, that it actually costs them less than 10 cents an hour to provide it to you. Indeed, the 2018 10-K filing with the SEC from Amazon shows its AWS business unit had $3 billion in operating income against $12 billion in sales, or a margin of around 25%.

But how big are the public cloud providers? Recent estimates suggest that AWS operates 1.4 million servers, Microsoft has over 1 million servers, and

1 A version of this article was originally published at The Art of Writing Software.

Google operates 2.5 million servers. At these scales, public cloud companies have leverage to get volume discounts from their supplies, but have also been forced to build automation and processes that are much more efficient—this is the *experience* part of *experience curve*. However, there's a markup, right? For your use case, there's a certain scale at which you can do it more cheaply yourself than it would cost to rent from them. How do we figure out that break-even point? Let's take a look at a 90% experience curve for servers.



We've added a higher curve that adds a 25% margin to the base cost curve (to represent retail pricing from the cloud vendor), and we've set the right edge of the graph at 1.5 million servers. If we draw a line left from the retail price at 1.5 million servers, we'll eventually hit the lower cost curve. This is the break-even point, and on our graph it occurs at around 345,000 servers. In other words, unless you are going to buy and operate at least that many servers, it is cheaper to rent them from a cloud provider!

But this is a conservative estimate. If any of the following are true, the break-even point is *even higher*:

- The learning curve is faster (lower) than 90%.
- The provider is operating at a higher scale than 1.5 million servers.

- The effective margin is lower than 25% (perhaps you can get a volume discount, or cloud provider competition drives the margin down over time).

Finally, the public cloud providers keep growing: as much 25% year over year by one recent estimate. This means they will continue moving further down the experience curve, perhaps faster than you can yourself even as your business grows.

Of course, cost is not the only consideration when deciding whether to use a public cloud provider, but it's an important one. And now, armed with an understanding of experience curves and economies of scale, you have a powerful heuristic you can use in your decision making.

Managing Network Transit Costs in the Cloud

Ken Corless

Executive VP for Technology, Offerings & Partners, DXC Technology

Cloud cost estimation is more complicated than ever.[1] Many development teams have a good idea of their computing and storage needs but are incapable of estimating their network needs. If you're ready to reduce your cloud spending, here are some places to start. (While these tips largely use AWS terminology, most apply to the other major cloud service providers as well.)

Make sure you are watching and measuring your costs. Use the AWS Billing and Cost Management dashboard as well as Cost Explorer. Consider third-party cost management tools, such as Cloudability or Teevity. Transferize is a product targeted specifically at optimizing cloud transfer/network costs. All of these tools have modeling capabilities—use them!

Set billing alarms (either in your cloud provider or your third-party tool), especially when you are just starting out or deploying brand-new workloads.

Don't use public/elastic IP addresses when you can use private IPs. This mistake is both common and costly.

Leverage a content delivery network (CDN) like Amazon CloudFront or a third-party CDN from a company like Akamai. You'll still be charged to move the content from your VPC to the CDN, but if you have a lot of common traffic (such as web pages), these savings can add up quickly.

Stay within an availability zone (or region) in places where you are not looking to improve availability. Needless region-to-region costs are a killer.

Leverage data compression. Whether for web pages or video files, compression makes a ton of sense when you pay by the byte.

1 A version of this article was originally published at LinkedIn.

Look at your interface topology. With the popularity of hybrid and multicloud approaches, where solutions running in the cloud are hooked to solutions on premises (or on other clouds), mapping the interfaces is helpful. The network topology will have performance implications as well, especially for real-time interfaces.

When replicating data, wherever possible, send only the changes in data, rather than forcing new, full copies.

By establishing a direct connection to the cloud, you can typically lower your costs. The pricing model is different (lower bandwidth charges, but you pay for active ports).

Reduce local processing. If you need to download data locally to your workstation, you may be able to leverage Amazon WorkSpaces as your workstation, thereby keeping data in the cloud.

Check your automated multiregion replications (like DynamoDB) for interregion network transfer charges. What data truly needs to be multiregion? In particular, terminate abandoned testing/development environments that replicate needlessly.

Have an effective tagging strategy. With instances tagged and cost allocation tags enabled, network charges can be associated with instances and ultimately the accountable application/solution teams.

Finally, place the accountability (and budget) for network charges on your application and solution teams. (Actually, do this for *all* nonshared infrastructure costs.) People act differently when it's "their" money.

Managing the Cloud Migration Cost Spike

Manjeet Dadyala

Authorized Google Cloud Platform Trainer

Costs exist with any technology modernization effort, and this cannot be understated when you or your organization is exploring or migrating to the cloud.

All too often, one of the value propositions pitched by cloud providers, consultants, and parties that participate in the ecosystem is that the cloud will save your organization money relative to current on-premises environments and workloads. While this is true at many levels, it is also very untrue for organizations that are beginning to explore, use, and migrate to the cloud while still maintaining on-premises infrastructure and workloads.

Organizations are often surprised at their cloud costs if they fail to plan for and manage the cloud migration cost spike. That spike is a key aspect of cloud economics and total cost of ownership (TCO). But what is TCO? It's a value proposition; it's a what-if scenario of your organization's use of the cloud based on several assumptions.

Failing to consider and adequately calculate all of the migration costs will likely result in a spike that not only exceeds expectations but also will be more costly to address after workloads have been migrated.

These costs often include the following:

- Discovery and planning
- Assessment(s)
- External consulting
- Network and infrastructure connectivity
- Proofs of concept
- Application preparation, refactoring, and readiness assessments
- Tooling, software, and licenses needed for the migration

- Deployment and migration activities
- Staff training and upskilling
- Turndowns of existing datacenters or colocation facilities
- Penalties for breaking existing software license agreements or location contracts

To understand and successfully manage a cloud migration effort, a cloud engineer in conjunction with other teams within the organization must put effort into analyzing these costs, their associated timelines, and their impact on a migration effort. This is especially true for organizations that will have dual running costs—costs associated with maintaining existing technology investments while incurring costs associated with cloud migration and modernization. Moving to the cloud is not an all-or-none event. Most organizations will not and cannot move their entire technology portfolio to the cloud in a singular effort.

Migrations can be scoped, limited, and optimized for level of effort, cost, risk, and speed. It's both a science and an art requiring each organization to determine a balanced mix of infrastructure, application, migration, people, and process costs. Perspective, knowledge, experience, and upskilling will enable the cloud engineer to not only expect the migration spike but also plan for it and manage it in accordance with what fits with the organization.

Damn It, Jim! I'm a Cloud Engineer, Not an Accountant!

Michael Winslow

Technology Leader

It was like any other Friday. Most of my calendar was blocked off with meetings that I had scheduled for myself so I could have time to take care of open items before the weekend began. Then something drew my attention to one particular email message.

The email was basically an automated report of our cloud spending sent from our cloud provider. Out of curiosity, I decided to take a look at the link title, "possible savings." What I saw was interesting to say the least:

- Size: 8xlarge
- Instances: 12
- Region: us-east
- Average Utilization: **0.27%**

Wait…that can't be right, can it? Did that say 0.27% utilization!? How much is that costing us?

After quite a bit of investigation and traveling from engineer to engineer to ask, "Is this yours?" let's just say that it turned out we were spending well into the six figures on glorified sandbox environments, all because we were completely unaware as a department of the financial implications of the cloud.

Once I realized how important it is to track exactly what you are paying for with the cloud, I became a bit obsessed. Not only did I want to know what we were spending our cloud budget on, I wanted to know if we as a team understood the most efficient ways to utilize cloud services.

Since that enlightening Friday afternoon, I have never ignored the automated email report from the cloud provider. Here are a few more things we've learned along the way:

For compute and storage, utilize reserved resources/instances.
: I'll be the first to admit that a lot of our early cloud adoption simply consisted of "moving our bare metal" to the cloud. That is a great way to gain understanding of the environment. But if you know that you have VMs that are going to live this way for years, cloud providers will offer you discounts. Do yourself a favor and get reserved pricing.

Moving to microservices? Think about network, storage, and monitoring costs.
: With all the great advantages of moving to microservices, the cost is often overlooked. Each new microservice increases the network traffic, as these chatty services need to communicate with each other. You will also find an increase in log sizes due to the additional tracing information that needs to be captured.

  In our case, we also found that some third-party monitoring solutions were slow to create a pricing model specific to the needs of microservices. They've historically charged by the "instance" or "node" since monolithic architectures rarely grew very large. But as our microservices count grew into the dozens, this billing strategy became expensive.

Always look for the /pricing page.
: Cloud services have different pricing models, based on the type of service. Some services will bill based on instance size, while others will bill based on traffic. This is important to understand. In some cases, you may even decide to make changes to the way your software is designed in order to take advantage of a cheaper solution.

As cloud engineers, we can no longer be oblivious to the financial impact of our decisions. Every call to an API, every log statement, every moment of computation has a real cost associated with it. This is an excellent opportunity to deliver measurable value to your company and your customers as well as to stand out as a senior engineer.

Effectively Monitoring Cloud Services Requires Planning

Scott Pantall

Software Engineer at infinicept

"Fast, cheap, or good? Pick two." This is known as the *iron triangle maxim* in project management, but many cloud providers will have you believe that if you just use the things they are selling, you can pick all three. They are not completely wrong, but they're not completely right either. If you can keep track of performance, alerting, and billing, you can make educated decisions to keep your application running just fast enough, just cheap enough, and just good enough to keep all your stakeholders happy.

Your salespeople and users want things to be fast, available, and reliable. The performance of your cloud-hosted services is integral to the satisfaction of these stakeholders, so it will be important to keep track of metrics that can result in slow or unavailable services. Before diving into the various ways cloud providers can offer these metrics, it is important to plan for what you want to happen in different scenarios. This is where cloud providers can really shine. If memory usage, disk space, or network traffic is higher than expected, what do you want to happen? Do you want more resources allocated? Do you want systems to restart? Do you want certain people to be alerted? Cloud providers can do any combination of these things, but you can take advantage of them only if you plan for them and use the provided tools to your advantage.

Your organization is likely attracted to the cloud because it is seen as cheaper than hosting things itself. Using a cloud provider means your team does not need to hire IT specialists to manage and purchase servers and network infrastructure to support your application. You just need someone who can understand how to take advantage of the provider's offerings. Unfortunately, one of the downsides to using a cloud provider is the potential for hidden costs to come up unexpectedly. Autoscaling resources is great for your users

but may result in unexpected costs if not managed and monitored. Take mindful action to really know the cost of hosted services and autoscaling options before you find yourself overusing them and running up costs.

Your development team takes pride in doing work that is complimented by users, profitable to the business, and (most important to the development team) easy to maintain and build on. Team members need to be aware of the complexities of your cloud services and to be able to easily read and understand performance metrics in order to troubleshoot immediate problems. They also need to easily understand performance metrics so they can prevent and respond to user issues that cannot be solved by just using a cloud provider's solution. Making a proactive plan to allow your development team to monitor performance and cost ensures they can do good work and keep things cheap and fast.

Monitoring and alerting tools for cloud services are just tools. To keep all stakeholders satisfied, it is important to have a plan to make the best use of those tools. Without a plan, your well-meaning use of these tools alone will not help you balance your stakeholders' desires for fast, good, and cost-effective software.

PART VIII

Automation

Principles, Patterns, and Practices for Effective Infrastructure as Code

Adarsh Shah

Independent Consultant

Infrastructure as code (IaC) is an approach that takes proven coding techniques used by software systems and extends it to infrastructure.[1] It is one of the key DevOps practices that enable teams to deliver infrastructure and the software running on it rapidly and reliably, at scale, especially in the cloud.

Key Principles

Two key IaC principles are idempotency and immutable infrastructure:

- *Idempotency* means no matter how many times you run your IaC and what your starting state is, you will end up with the same end state. This simplifies the provisioning of infrastructure and reduces the chances of inconsistent results. Idempotency can be achieved by using a stateful tool with a declarative language, like Terraform, where you define the desired end state and then it is Terraform's job to get to that end state. If it can't, it will fail.
- *Immutable infrastructure* means instead of changing existing infrastructure, you replace it with new. By provisioning new infrastructure every time, you are making sure the configuration is reproducible and avoiding drift over time.

1 A version of this article was originally published on the Adarsh Shah's website.

Principles and Practices

The following are important IaC principles and practices:

Source control
: Everything should be in source control—even a script that you run occasionally—and should be accessible to everyone in the company so anyone can look at code and understand what is going on.

Modularizing and versioning
: IaC helps with maintenance, readability, and ownership across various teams. It also keeps the changes small and independently deployable. In organizations that have separate teams such as networking and security, it might make sense to separate various layers of your infrastructure and give ownership to appropriate teams to allow better control.

Documentation
: With IaC you should not need extensive documentation, but some is still essential. Good-quality, up-to-date, and easily available documentation helps not only the team that is maintaining IaC, but also its consumers.

Automated tests
: Adding automated tests for your IaC will help you find issues faster and earlier in the cycle:

  - Run linters and other *static analysis* on your IaC to ensure adherence to team and industry standards.
  - Since the IaC tools are declarative, *unit testing* is usually not needed. In some cases, though, it might be helpful (for example, when you have conditionals or loops).
  - For *integration testing*, provision resources in an environment and verify whether you have met requirements. Remember not to write tests for things that your tool is responsible for, especially if you are writing declarative code.
  - Running *smoke tests* with dummy applications helps in verifying that you can deploy the types of applications you will be running on your infrastructure. Use a dummy application to test scenarios that your real application will face but that are not configured for production.

Security and compliance

Making sure the infrastructure you provision is secure and compliant is important. Using robust identity and access management for your IaC helps the cause. Techniques like role-based access control reduce the overall attack surface by giving just enough permission to perform the operation required. Use a reliable secrets manager for any secrets needed by your IaC. Running security scans after provisioning/changing infrastructure in a lower environment helps avoid security issues in production and ensures that security best practices are followed. Companies with compliance requirements like the Health Insurance Portability and Accountability Act (HIPPA) and the General Data Protection Regulation (GDPR) should use *compliance as code* to automate compliance verification.

Automating execution from a shared environment

All the preceding steps should be brought together to execute IaC with appropriate checks in a certain sequence to provision infrastructure with confidence. There are two options for this:

- An *IaC pipeline* brings all the steps together by using a pipeline tool like CircleCI, provides visibility to everyone dependent on the infrastructure, and notifies on a failure.
- *GitOps* extends IaC and adds a workflow (pull request process) to apply a change. It could also have a control loop that verifies periodically that the real state of the infrastructure is the same as the desired state.

Red, Green, Refactor for Infrastructure

Annie Hedgpeth

Senior Cloud Automation Engineer at 10th Magnitude

When a company decides to move to the cloud, I can guess what its main motivation is: velocity! I'm sure you've been there. You plan a new project and begin implementation. You get excited about seeing it come to life as you deliver new features one after another. Pretty soon, however, the feature requests start piling up, and you slip behind schedule. But what changed? The growing complexity of your project forces you to spend more time testing your changes than you spend on actually making the changes. To meet your stakeholders' expectations, you begrudgingly accumulate tech debt and promise to start integration testing as soon as you get a free cycle. Still, you're frustrated that nothing works, and your team is burning out. That free cycle never comes.

Unfortunately, this is an all-too-common scenario in cloud infrastructure development—but I promise you that there is a way to develop your infrastructure that is less stressful and more fun. This way is called *test-driven development* (TDD), or *red, green, refactor* (which I prefer because of the reassurance that a green passing test gives). TDD means that you will not write one line of code without first writing a test for it. The workflow is to write a test, run it, watch it fail (red), write the code to make it pass (green), and repeat (refactor).

Here's a simple example. Say you need code to spin up an Ubuntu machine in Azure. Before you write the code that creates the machine, you first write code that checks Azure for its existence. You have a plethora of tools at your disposal: PowerShell, the Azure CLI, InSpec-Azure, and many more. Your test fails because that machine doesn't exist yet, so you now write the code to make that machine exist, again using your preferred tool: Terraform, Azure Resource Manager (ARM), etc. Now run the code to create the machine, and then run the test to see if it's there, and watch it pass. Rinse and repeat until your entire environment is built.

Why go through all the trouble, you ask? Well, we like and trust our teammates well enough, but how do we know that they're testing their code? We don't unless we build testing into the development process. As a project grows in complexity, manual testing becomes more cumbersome, and bugs slip through the cracks. Getting things to work starts taking longer, and you face burnout. However, if you have been building tests at the same time as building your infrastructure as code (IaC) through a TDD practice, then you not only already have the tests, but you will have enjoyed your development process a lot more! Instead of spending excessive time trying to get one gargantuan thing to work, you enjoy seeing small passing tests all day long. You can now dig the integration testing out of the tech debt pile and use it to create a simple continuous integration (CI) pipeline that will create a test environment of your IaC, run those integration tests, and have a gate that doesn't allow code to be checked into source control unless all of the tests pass. Now you're not stuck with bad code that you don't know who broke. You've successfully tricked your team into making the right thing to do the easy thing to do, and everyone's development process is more enjoyable.

Moving fast is fun. Seeing the implementation of your plan come to life is thrilling. Writing code and seeing it build and configure infrastructure in the cloud is very satisfying. But if you don't use integration testing—yes, even for cloud infrastructure provisioning—then you will be slogging uphill the whole time. As evidenced by DevOps Research & Assessment (DORA) research, TDD is at the heart of velocity, success, and enjoyment of our work. It cannot be underestimated or put off. If you're an engineer, embrace it, and if you're a leader, champion its cause. It will pay dividends.

Automate or Not-o-Mate?

Judy Johnson

Software Engineer at Onyx Point

Ask a group of engineers how to DevOps, and many will say, "Automate all the things!" That's a great answer that definitely covers a lot of the DevOps process, but it is not all DevOps is about. Here, I discuss why automation is important, how it relates to DevOps, and when automation may or may not be the right tool for the job.

Let's take a step back and concentrate on automation. *Why automate?* There are many reasons; among them are to save time, ensure consistency, reduce the chances of human error, save the cost of a human performing the task, or allow the process to be part of a larger automated process (i.e., CI/CD). *What do we automate?* We automate tasks that are repeated often, are error prone, or need metrics and status collection, as well as ongoing processes such as testing and deployment. *How do we automate?* We use scripting or computing languages; configuration management tools such as Puppet, Chef, Salt, and Ansible; continuous integration frameworks such as GitLab CI/CD, Jenkins, and Travis; cron jobs; APIs; and variations on these themes.

Now let's talk about the DevOps phases: plan, code, build, test, release, deploy, operate, monitor. Many of us are automating most of those phases already. In my opinion, the cool thing about DevOps is that if you have completed any phase, you have already made your task simpler.

So you have a process to automate—what do you do? The planning phase is arguably the most important and the least automatable. To see what you are doing, perform it manually, and then document what you have done. You've already simplified life with instructions! Next, code or script your process, solidifying the steps and building a product. The test (and peer review) steps allow you and others to understand what's been done from a slightly different perspective. Automations of the release and deploy phases are time-savers, and can be reused as the product is continuously improved. In production, continue to operate and monitor, as continuous feedback is important.

Sometimes automation may not be the appropriate tool for the job—for example, for code reviews. Yes, automated checks can add consistency, check coverage, and reduce errors, but a human eye is still important in most situations. Your process may need a manual step, such as checking interim results or metrics, or entering a password. Most importantly, there is always a need for human creativity. You may have your entire process automated, but there are always improvements, bug fixes, and new requirements to implement.

Another hindrance to the process is the naysayers—people who think their job will go away if they automate too much of it, or are just not comfortable with the tools. To these people, my response is that you are not taking away your work; you are giving yourself time for more challenging and meaningful work.

My favorite thing about DevOps is the feeling of community. You will likely reach a point where your product is fully automated and you can move your creativity in another direction, but it is important to ensure that every step of the way there involves interaction, communication, knowledge sharing, and documentation. And feedback—you will definitely have lessons learned to apply to your next creative endeavor. To make an analogy to my second-favorite thing (after DevOps), baking: even a perfect recipe needs a little human touch, and perhaps a slight tweak, each time you make it. So automate away, but ensure that the human touch remains part of your process.

Beyond the Portal: Manage Your Cloud with the CLI

Marcello Marrocos

DevOps and Cloud Advocate

How long does it take to set up a server from scratch? Buy hardware, assemble, install the operating system, and it's done. That's how it used to be.

With the cloud, you can just log in to an administration portal with a friendly and intuitive graphical user interface (GUI), and with few mouse clicks and configuration settings, your server is ready. In less than five minutes, you can remotely log in to your newly created server.

So far, so good. But imagine that you have to create 10, 50, or 100 new VMs. Using the GUI gets impractical. The task that took you five minutes will keep you busy for over eight hours if you need to create 100 new VMs manually, repetitively. And for a significant part of the five minutes it takes to create each one, you will be doing nothing but waiting for what you hope will be a success message. Moreover, there is the risk of making a mistake when typing the name or selecting the image.

But don't worry, there is a practical solution for this: the command-line interface (CLI). Every major cloud solutions provider has a CLI, for several platforms. The structure of the commands may vary, but will often contain the provider's initials, the resource (subject), and the action (verb), plus the additional parameters required for the operation.

For example, the following commands are used to create a virtual machine:

Microsoft Azure

```
az vm create
```

Amazon AWS

```
aws ec2 run-instances
```

Google Cloud

```
gcloud compute instances create
```

Note that those are only examples, and additional parameters, such as the resource name, size, and image type, might be required for the action to complete successfully.

Apart from the excellent documentation provided by the cloud providers, an extra tip is the parameter `help`, which will give you the flags for the specific resource and action, making it easy to check the required settings and different options.

Using the CLI by itself will not entirely solve the problem of creating 100 VMs. It will help to keep consistency on a few parameters, but you will still be risking mistyping the names of the new VMs and will still need to wait the five minutes for the command to complete for each item before running the next one.

So, the next step is to incorporate the CLI commands into a script. This is where the automation starts. With scripting, the sky is the limit. In this scenario, imagine that you create a loop in which you increment a number variable and use it to define the name of the VM. No more mistyping risk. Moreover, the commands to create each VM will be executed one after another, without the need to wait for one command to finish before manually running the next one. You just execute the script and come back eight hours later with all the resources created.

That's just a simple example, and more complex scenarios can be explored. You can create conditions, log results, and execute other commands depending on the results of previous commands, for example.

This doesn't mean you will never use the portal again. For particular tasks, the portal is my primary interface. For instance, I once needed to create a new deployment slot for a web application on Azure. And after creating one, I had to download the publishing profile to get the credentials and fill in my deployment pipeline. I did this in the portal—punctual, fast, and accessible.

A couple of days later, I was asked to do the same for another system. But this system had 54 web applications. No way would I do it manually 54 times! It was a matter of 10 minutes to research and create a script to iterate through the web applications, create a new deployment slot for each, and save the publishing template file into a local folder. The execution took less than two minutes.

Besides these particular tasks, CLI scripts can also be used on automated CI/CD pipelines for specific actions; for example, to create on demand a web application instance to host the code that you just built. Keep these scripts in source control, and build a pipeline that consumes them when they change, and you've started moving toward infrastructure as code—but that's a subject for another article.

Meanwhile, give the CLI a try. Understand the command structure, practice, and get comfortable with it. You won't regret it.

Treat Your Infrastructure like Software

Zachary Nickens

Site Reliability Engineer at Woolpert

Infrastructure is important. Infrastructure and application code are equally critical to success as a cloud engineer. Most engineers either choose the correct runtime environment, or iterate through runtime environments until they find the appropriate one for their application. How you provision, deploy, and recover whatever infrastructure you use is critical to choosing the appropriate runtime. Most cloud engineers love designing, architecting, developing, and deploying applications. Error reporting, debugging, logging and log aggregation, and alerting generally are easily baked in when working on a major cloud platform or working with common toolsets.

One of the greatest advantages to working in the cloud is the plethora of managed services and tools readily available to meet those challenges. Cloud is awesome! That's a knife that cuts both ways, however. Managed services that are easily turned on can be easily and accidentally turned off. A managed database or a function running on serverless compute can be inadvertently dropped. And if you are provisioning and deploying those resources by hand or via shell scripts, they can introduce unnecessary toil into your downtime recovery and remediation strategies. And *nobody* wants toil.

Using managed services is a great strategy. Treating managed services as infrastructure and defining them using declarative and idempotent tools is an even better strategy. Define and declare your infrastructure as code, check that code into your version control system of choice, and peer review that code before changes make it into your live systems. This will save you downtime, heartburn, and headaches.

There are a wide variety of infrastructure-as-code patterns and tools. The most basic form of IaC is to simply write shell scripts to create your infrastructure. This method is not optimal. Scripting infrastructure provisioning is imperative and lacks the benefits of parallelized execution and dependency management—it's just a scripted version of manual

provisioning/deployment. Maintaining and debugging scripts introduces unnecessary toil into infrastructure. To avoid introducing all that potential toil, we can use IaC tools and methods that are declarative *and* idempotent.

Each of the major public clouds offers its own IaC tooling. Amazon provides AWS CloudFormation, Google has Cloud Deployment Manager, and Microsoft has Azure Resource Manager. These IaC tools offer degrees of declaration and idempotency, but they all work only in their respective public clouds. As multicloud and hybrid cloud approaches become more common in the industry, this isn't the direction we want engineers and SREs moving in.

Good IaC uses idempotency to compare your code (desired state) and your current state and to identify drift. Tools like Terraform present this comparison and then give you a ready-to-go remediation plan to bring your current state back into harmony with your desired state. Automating away infrastructure drift with IaC tools is essentially a superpower for cloud engineers. Cloud engineers who design, provision, deploy, and remediate efficiently and effectively build reputations as reliable and capable of delivering.

PART IX

Data

So You Want to Migrate Oracle Database into AWS Cloud?

Asha Kalburgi

Database Engineer and AWS Developer/DBA

Cloud databases are increasingly popular for reducing IT complexity and operating costs as well as dependence on specialized IT teams. Oracle Database is great, but it's also one of the most expensive and complicated options, and it comes with proprietary code elements.

When thinking about moving your Oracle workload to the cloud, consider that users have three ways to operate Oracle Database in the AWS cloud:

EC2
: Use the AWS compute service along with the block storage service Elastic Block Store (EBS) to run traditional self-managed Oracle databases in AWS. This requires a significant level of expertise from the IT team.

RDS
: Use Amazon RDS for Oracle, which is a managed relational database service. This falls under the database-as-a-service (DBaaS) deployment model. AWS takes care of installation, hardware setup, configuration, provisioning of storage, and network setups. It also takes care of server maintenance, software patch management, version upgrades, and backups. Scalability/elasticity, reliability, and high availability can be achieved easily.

Aurora
: This is an RDBMS developed, designed, and optimized by AWS for its cloud infrastructure and its DBaaS deployment model.

Migration of Database

Moving databases is the trickiest part of cloud migration, which often requires downtime and reworking of the schemas; hence an iterative process is better. Here is a brief checklist to follow:

1. Assess the current environment/database. This is an important step in the overall migration process. Determine the features of the Oracle system currently in use; for example, partitioning, OS-level process control, and size of the database. This will help in determining how much storage you need in the cloud and establish cloud compatibility. This must be done at the schema for each of your databases. Depending upon the results of this assessment, the migration complexity will vary.
2. Choose the deployment model: self-managed cloud database or DBaaS. The easiest choice may be self-managed EC2 servers, as users retain full control of the underlying OS, while a switch to RDS/Aurora is the most involved method to replace an Oracle database. With a self-managed model, EC2 users retain control and the ability to customize, though it carries additional management and overhead costs for database administrators (DBAs). When users choose the DBaaS model, DB maintenance tasks are handled by AWS, reducing the dependency on availability of skilled resources for daily operations.
3. Choose the target RDBMS technology. If the target RDBMS is not Oracle, migration becomes a two-step process. The first step is to convert the database schema with the AWS Schema Conversion Tool (SCT). Then, you copy the data to the new data store using the AWS Database Migration Service (DMS). AWS provides a set of Database Migration Playbooks that guide you through the process, including configuration settings, and present best practices to streamline a successful heterogeneous database migration.

Helpful Tools

Two tools, already noted in the preceding checklist, are helpful:

AWS Schema Conversion Tool
: SCT will generate a conversion report that identifies the issues, limitations, and actions required for the schema conversion. This report helps with assessing the complexity of an RDBMS conversion, and whether a DBMS can be converted. The tool also generates target schema conversion scripts to be applied before running DMS. These will do any

necessary code conversion for objects like procedures and views. Not all objects can be converted, and SCT reports can help identify those clearly.

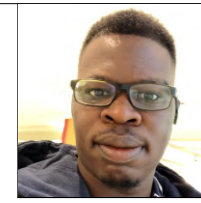
Database Migration Service
: After the schema is converted and updated for Postgres RDS/Aurora, the DMS can load or copy data into the target database. You can arrange for an ongoing replication from source to target with minimal setup and configuration changes on the source database. Validations and CloudWatch logs help debug any issues in migration of data. Following migration and validation, applications can be switched to a migrated database with minimal downtime.

In my opinion, although the SCT and DMS tools simplify the most tedious tasks in the process of data migration, they are nowhere near complete solutions. Heterogeneous migration does involve significant development efforts and must be implemented after careful consideration.

DataOps: DevOps for Data Management

Banjo Obayomi

Senior Software Engineer at Two Six Labs



As data becomes more ubiquitous in software projects, a culture has emerged around providing processes to manage data with a DevOps mindset. *DataOps* is a new paradigm that invokes DevOps principles and applies them to managing data to create value. As a cloud engineer, being able to enable your data team to work with the same efficiency as a software team is important, thanks to DevOps principles. Looking at problems with a DataOps lens will help ensure that your data team can deliver value to your end users. Here are three things to focus on when incorporating DataOps ideas into your processes.

Reproducible Data

As cloud engineers, we are used to spinning up and spinning down environments with a click of a button, but when it comes to collecting and storing data, the process is not always so clear-cut. Questions like "What if we can't get data again?" or even worse, "We don't know how we got our data" should scare any engineer. Our data pipelines should follow the same principles we use for tracking, deploying, and updating servers. This approach will help reduce the fear of not being able to get data again, and remove bottlenecks caused by barriers to sharing and collaborating with our data.

Analytics as Code

As cloud engineers, we have created numerous YAML files depicting every asset deployed into the cloud, and we understand how all of the pieces fit together. When it comes to data analysis, we often rely on a dashboard program that's completely siloed away in a Docker image. What if we took the "as code" mindset and embedded analytics in a similar fashion? Analytics can be thought of as code and configuration that describes actions upon data in order to deliver insight to a user. Having a robust reproducible data pipeline allows for an analytics-as-code process to emerge.

Data as a Platform

With a robust data pipeline and interoperable analytics, naturally the next step will be to build solutions based on the insights gained from the data. As a cloud engineer, one of the main goals is enabling the team to build solutions in a fast, secure, and scalable way. When it comes to data teams, these principles are often forgotten, and nonscalable solutions are used to solve problems. By offering data as a platform, we take the same methodology of allowing developers to spin up their own environments to quickly prototype ideas to data professionals that need that robust pipeline and analytics to perform value-adding tasks.

As a cloud engineer, you're an integral part of any team. As DevOps has become more synonymous with the cloud, it's time to start thinking about the next paradigm shift. As machine learning and AI continue to grow in adoption, data will become increasingly important in many organizations. Your role will evolve to enable data teams to continuously deliver solutions, and that will require a DataOps-focused perspective.

Data Gravity: The Importance of Data Management in the Cloud

Geoff Hughes

SRE Senior Manager at NetApp

Data has gravity. Data gravity is akin to a black hole; data has a relentless pull. Starting cloud application design with data management allows for cost optimization and scalability. To understand data gravity, and begin to explore data management, consider three principle areas: data availability, disaster recovery, and data retention.

Data Availability

Data unavailability makes for a dismal user experience. What happens when you choose instance storage instead of persistent storage, and you lose an instance? Does your application handle that with application replicas on different instances? Have you chosen the right storage performance tier so that a performance issue does not present as data being unavailable?

Does the design account for data availability across multiple availability zones in a region, and potentially across regions? If you use a platform like Kubernetes, are you implementing the proper data availability to worker nodes securely? If a data store exists in only one availability zone, and that availability zone is lost, is your application still viable? Maximize data availability by design.

Disaster Recovery

Disaster recovery is what enables you to run your application if its primary region is impacted or becomes unavailable. Design the application to be multiregion from the start, and disaster recovery becomes an inherent capability of the platform. However, a multiregion design can add significant expense and complexity. As an alternative, consider a primary region and a disaster recovery/failover region.

How do you define your recovery time objective (RTO) and recovery point objective (RPO)? Do the storage or application technologies you've chosen support these RTO and RPO definitions? How often do you execute disaster recovery exercises (monthly, quarterly, annually?), and are those exercises manually executed or automated? Are the exercises successful?

Don't underestimate the role of data gravity in a disaster scenario. How frequently is data updated, changed, or added? What is the latency between regions, and can you replicate the data between regions within the required time? Data that is fairly static has less gravity than data that is highly dynamic.

Documentation is critical in order to have successful disaster recovery; don't assume any level of expertise when putting together a disaster recovery workflow.

Data Retention

Data gravity increases with the amount of data retained. The speed with which an application performs can be dramatically impacted by the amount of data to process. Does the application require retaining all data forever? Can a window be defined where the data is most useful? Fifteen months or five quarters of data is a suggested starting point. Depending on your industry (financial institutions come to mind), active data retention may be longer.

In addition to primary storage, backup and archive data retention also need to be considered. Backup and archive should be considered separate use cases. Backup data retention defines the ability to recover from a mistake. Limit the backup retention time frame by considering how far you will want to roll back your data. Do you want to roll back to a state five hours ago, a day ago, or a month ago? Somewhere between 8 and 35 days is a suggestion of where to start. Don't neglect data backup frequency when determining the time frame; if you take only one backup a day, you have already defined your RPO as no less than one day. An archive data retention policy should be aimed at long-term retention and support business objectives like auditing (again, financial institutions may have well-defined long-term retention policies that extend seven years or longer). Understanding the use case for archive data will help define the best approach for implementing an archive data retention policy and solution.

Data Gravity

Data has gravity. As you develop your cloud applications, consider the three principles of data availability, disaster recovery, and data retention. Don't let data gravity suck your application into oblivion; manage your data from the start.

PART X

Networking

Even in the Cloud, the Network Is the Foundation

David Murray

Principal Solution Architect at AWS

Despite all the hype and marketing focus on AI/ML and serverless, a huge portion of cloud customers still struggle with networking. Unfortunately, too many networking people come to the cloud and feel they have to quickly diversify their skills and move toward the latest buzzword technologies. This leaves many engineers in a jack-of-all trades situation as they attempt to shed the core skills that got them to where they are now.

If you are a network engineer, your skills are very much transferable to (and needed in) cloud computing. Become the cloud network expert and see how many times you end up as the center of the meeting when your company or customers are designing their cloud architecture. Whether they are looking at an all-in migration, hybrid cloud, or multicloud solution, none of this works without a solid network architecture.

Build upon your deep understanding of infrastructure and add the skills needed to automate the network in the cloud. Learn infrastructure as code and the ability to deploy your architectures quickly, in a repeatable manner.

You will soon find that everything you enjoyed about working as a networking engineer is still available to you in the cloud, but at scale! Now you can design and build globally distributed networks and deploy these in minutes. You can help your company or customers open up new revenue models when you can showcase how they can quickly expand their technology stacks to other geographic regions, and you can be the one to make this happen.

When you want to diversify, use your networking skills to your advantage. Cloud-based solutions like containers, service meshes, the IoT, CDNs, and edge computing all require deep networking knowledge. Let's take a look at Kubernetes as an example. The heart of what makes Kubernetes so brilliant is the network. If you look below the surface, you will see that from ingress

controllers to container network interfaces to service discovery, these are technologies that you are already very familiar with. You may also find that the people who are running the Kubernetes clusters are not traditionally networking people and could use your expertise.

Check out the latest trends in network technologies and understand how they may be applicable to customers in the cloud. 5G will be the driving force behind next-generation application architectures and technologies such as augmented reality. Staying ahead of these trends means that you can bring new perspectives to conversations that your colleagues are having. It allows you to see around corners, and this skill is transferable to any career path you will ever choose.

The key point is that you should not ever feel that your skills are no longer required in the cloud or that you need to stay working in traditional networking. You will be very much welcomed into the cloud community and will find a voice quickly by showcasing what you already know. Embrace the fact that you are an expert in infrastructure. Don't feel that the latest buzzword is the future of the cloud. Do, however, be curious about the latest technologies; explore them, understand their benefits, and then go a little deeper to see how they work at a networking layer. Then be the expert from the network perspective and get involved in the design discussions. You will find that becoming known as a cloud networking ninja pays huge dividends for you and the companies you work with. Be the enabler, and the next time you see somebody wearing the cool swag, be proud that you are the one wearing the Run BGP shirt.

Networking First

Derek Martin

Principal Program Manager for Microsoft Patterns and Practices

Far too often, organizations forget that the cloud, for all its serverless, PaaSified glory, still relies on actual network routing to get the job done—and you, as a cloud engineer, need to understand the basics to be successful. Each application that you deploy into the cloud will have multiple independent running parts with a network binding them together. How do you stay secure? How do you maintain availability and resiliency in the face of disaster? Microsoft Azure's networking primitives dazzle even the most hardcore networking admins, but without careful planning, you and they can be surprised when the unexpected happens.

Traditional three-tier application designs with VMs sitting in each of three independent networking zones can still be achieved in the cloud, but more modern approaches can keep your application stack secure without the overhead. These include *network security groups* that get applied to a subnet-based design and *application security groups* that get applied across applications and their tiers. *Azure Private Link* allows public PaaS services to enter this walled garden, but can be complicated by needs surrounding transit routing, peering limitations, and traffic shaping. If you are bringing the traditional three-tier application design to the cloud, keep this most important aspect in mind: unless everything in your stack is a VM, forced tunneling will cause a great deal of headaches, and there are more modern ways to secure your applications—like Azure Firewall, Security Center, and Advisor.

Hybrid application stacks that rely on PaaS or serverless components including Azure Functions, Web Apps, Kubernetes Service, and Database for MySQL can operate using traditional methods, with limitations. Challenges start to crop up if one of the application tiers remains on premises, however. These complexities can often be avoided by moving that remaining tier into the cloud as well or by leveraging App Service or Integration Service Environments. If you're using Azure Kubernetes Service, make sure you understand the intricacies of your ingest controllers, secret managers, and—if you

are using container instances for pod scale-out—networking considerations on crossing subnet boundaries. While they are supported, do not rely on large numbers of Hybrid Connection endpoints to connect your tiers, however. Use ExpressRoute with peering enabled instead for production, enterprise-grade connectivity to on-premises. Finally, *never* expose a VM's open port to the public internet without leveraging DDoS standard protection.

Even for cloud native applications—those that have no on-premises or VM-based needs—you still need to consider networking. From a security standpoint, you want to work with each Azure SKU to make sure that connections are limited to just the correct endpoints. You also want to protect your application's public endpoints behind Application Gateway within a region and Front Door and/or Traffic Manager to remain resilient across multiple regions. If your application (or part of it) is an API, leverage Azure API Management, which now also includes a serverless/consumption-based mode. Other considerations include the need for internal load balancing, the limits of the serverless modes (yes, there are limits), and scaling up/out appropriately.

In a disaster, the first thing that will need updating or controlling is the network. Have plans in place and practice shifting load around multiple regions. Never take a single region dependency and make sure your data tier supports the ability to move between regions with RTOs/RPOs that comport with your organizational needs. Leverage Azure Front Door and/or CDN to stage your applications' static assets at the edge, as close as possible to your users. In addition to protecting against regional failures, these steps will allow you to shift load and maintain proper performance of your application in the event of regional internet congestion or failure, and to keep your traffic closer to the end users.

Handling Network Failures in the Cloud

Shayon Mukherjee

Infrastructure Engineer at Loom

In the era of cloud computing, network failures—especially transient ones—are a given. These failures come in many forms and can originate from servers, routers, load balancers, connection pools, software applications, human errors, and, of course, the DNS. Writing software applications for the cloud in a distributed system environment therefore requires an added degree of care and a resiliency mindset. Use this mindset to incorporate practices during software development that will allow your applications to withstand these failures without disrupting customer experiences.

A common way to handle network failures is through the use of timeouts, retries, and retries with backoff and jitter. As a cloud engineer, it should become part of your DNA to enforce these practices when dealing with network connectivity or similar communication protocols over the internet—because things always go wrong in production.

A *timeout*, in simple terms, refers to the maximum time allowed for a connection to sit idle. Lack of time-outs when combined with connectivity issues to another service often leads to increased latency and resource exhaustion. In such a scenario, a client and server are waiting on a request that may never complete, resulting in a nonoptimal customer experience. As a cloud engineer, you should put timeouts on all network call in order to reduce the blast radius of failures.

Many modern-day applications and clients provide the ability to implement timeouts on network calls. The hard part is figuring out the appropriate time limit. What may work for a DNS resolution may not work for a database query. A general rule of thumb is to look at past request duration (latency) for the service(s) involved to find an acceptable baseline for optimal customer experiences. This process often requires a few iterations to settle on a figure that is sustainable.

Retries, as mentioned previously, are a nice way to combat transient failures. Because of the nature of requests in the cloud, a subsequent request after a timeout often yields success. When a client receives an error response for a timeout, it is the responsibility of the client to retry. Retries in nature can be belligerent, such that using retries without upper bounds is a recipe for DDoSing your own systems. As a cloud engineer, unbounded retries should tingle your spidey senses. Additionally, putting retries in place for every network call in the stack may not be wise. Pick your battles.

Backoff is a technique for performing retries gracefully, without overloading or burning out your backend systems. A simple way to perform retries is by adding a delay between calls. This approach is called a *linear backoff*. While this is easy to implement and can handle transient failures in a majority of cases, it does not help when a downstream service is impacted for a prolonged period of time, as the retries sent at a fixed rate will continue to overload the service.

Exponential backoff is a less aggressive form of backoff. As the name suggests, with this approach the delay between each retry increases exponentially until the request succeeds or an upper-bound retry limit is hit. This is a more graceful strategy because it avoids overloading downstream servers, which can result in resource starvation.

Backoff with jitter is another beneficial technique. While exponential backoff allows you to spread the retries more scientifically than linear backoff, it still leaves the backend systems open to request bursts on every retry, potentially leading to resource exhaustion. To deal with this, we can add jitter to our backoff strategy. In other words, we introduce randomness to the retry intervals. Instead of retrying at a fixed interval (exponentially), each client will retry at varied intervals. This is especially beneficial when a large number of clients are distributed and are coordinating with a specific set of central backend systems.

Lastly, whatever strategy you settle on, be sure to test your settings in production. :)

PART XI

Organizational Culture

Silos by Any Other Name

Brittany Woods

Lead Cloud Automation Engineer at H&R Block

The principles of DevOps have taught us that we need to increase velocity and innovation. To accomplish that, we want to "automate all the things" and use the best tools available. We also want to eliminate the silos that for so long have hindered our industry. The ultimate aim is to give teams autonomy to innovate and drive business outcomes.

With the emergence of cloud platforms, we are no longer stuck with servers that have to live in the company's physical datacenter. Nor does the power to build servers lie solely with systems engineers, relying on the time-consuming process of submitting requests and waiting on setup.

As engineers, cloud platforms have provided us with a more accessible method to hone our craft. This method has allowed us to learn the ever-growing list of emerging technologies and take that information back to our companies to drive the next phase of innovation. We do not need to build up a datacenter in our bedroom closet, though many of us still do. Barriers to entry, in this regard at least, have been lowered.

So, we're all cloud engineers now. We live at the intersection of application development and system engineering. Our job has changed once again, as it so often does. We all must understand hundreds of tools and technologies, advanced networking principles, security, CI/CD, application development…the list goes on and on.

The pressure to keep up with the latest list of technologies and get on board with utilizing them is higher than ever. When the rapidly developing innovation of your competitors is added to the mix, that pressure is increased tenfold. Cloud engineers are the new humans-of-all-trades. Yet, buyer beware. The emergence of all of these easy-to-use and easy-to-license tools has led to the latest in silo-forming fashion: tool creep. Every cloud engineer should be aware of how the infliction can cause you to quickly go from practicing DevOps to reestablishing silos in your organization.

Suppose that you have a team with a fully implemented solution, and another team is getting ready to solve the same problem but in a different way. These two teams are now forming silos of work that only they understand. As responsible cloud engineers, our expertise is vital to keeping these silos from forming in the first place. It's incredibly easy to fall victim to the glamour of newly available tools, but we have to constantly gauge whether implementing these tools makes sense. What will using this tool buy us? Is there something to gain? Is this tool uniquely equipped to handle our use case? Does it do the same thing as what we already have?

We must not only understand the tools available to us but also be aware of the functions and possibilities of other tools used within our organizations. This process is vital to recognizing the warning signs of tool creep. The adage "just because you can doesn't mean you should" really rings true here. As cloud engineers, we do not need to be experts in every tool—a notion that has carried over from decades of IT practices of old. However, thanks to our intimate position at the intersection of building and implementation, we have knowledge that can make everyone's lives easier down the road.

We need to make a concerted effort to guide our organizations down a path that sets them and everyone else up for success, not one that drags us all toward an imminent failure of death by tools. Again, we are the ones best equipped to answer these questions and to make recommendations as a result. Our positions and expertise are unique. Never forget it.

Focus on Your Team, Not on the Cost

Guillaume Blaquiere

Scrum Master and Lead Developer at Veolia

Public clouds come with a new capability: the real knowledge of cost. In my company, this generated fears: "Whoa, it's expensive!" No, it isn't—but the cost is known, visible. In an on-premises environment, it's very hard to know the real cost per service, especially the human cost or the cost of mutualized resources (network, storage, hosting, etc.).

When you choose a technical solution, it's important to take into account all the pieces of the project—and first of all, your human resources. Google Cloud Functions, AWS Lambda, or other FaaS solutions may seem cheaper because your use case fits them well. But is your team able to develop functions in the available languages and within the existing limitations?

The scalability and simplicity of these serverless solutions makes them popular, and companies have built entire new stacks on top of them. Some projects have thousands of functions, and it's hard for a newcomer to understand the impact and the role of each one.

For a lot of teams that move to the cloud, splitting their monolith applications into microservices is a real challenge in terms of design, documentation, and organization. If you consider a simple microservice with four HTTP verbs (GET, POST, PUT, DELETE), FaaS architecture splits it into four parts. Functions are simple, but hard to consolidate and test, and it's difficult to keep consistency among them. Code factorization and reutilization also can be a challenge, and the tests are often redundant.

In addition, the current limitations, like the difficulties in using private libraries when deploying or loading third-party binaries can be a real challenge when you start a FaaS project. Sometimes you have to reinvent the wheel to cope with the existing constraints.

Do you really know the cost of your FaaS choice? Here are some of the human considerations:

- What is the impact of valuable/senior employees resigning because they are not comfortable with the new stack?
- What is your training session budget?
- What is the impact of having no productivity during the training session?
- How do you manage the lack of code quality, reliability, performance, and productivity because of the new stack?

These costs, most of the time, add up to far more expense than the cost of the cloud provider.

In addition to this human aspect, there are other potential costs. What will be the cost of refactoring to achieve portability? What will be the cost of having *x*% more bugs because the software is difficult to test or the team lacks experience in a new language?

Before you start using an online pricing calculator, be aware of your teammates' capabilities, wishes, and skills. Start to think about containers and portability. Be smart before being a cost killer!

Cloud Engineering Is About Culture, Not Containers

Holly Cummins

Development Lead, IBM Garage

What makes an application a *good* cloud application? The 12 factors are a good start, but they address only a small part of the challenges of behaving well in the cloud.

There's a story in my team of a bank that asked for help to convert an aging COBOL application to cloud native microservices. I imagine that the bank's goals were to become more modern, more agile, and better meet customer needs—but its release board met only every six months. Another client developed an architecture with 68 microservices, but was a bit uneasy about how they'd interact. To ensure that changes in one microservice couldn't negatively impact other services, the client designed its CI/CD pipeline to ensure that all 68 services could be deployed in a single atomic step. This allowed them to run a lengthy user acceptance testing (UAT) exercise before doing any releases.

There's nothing *wrong* with doing releases this way, and for many years, this kind of release cadence was the norm in our industry. If the 68 microservices were distributed but not decoupled (an easy trap to fall into), releasing them in a block and carefully testing them as a unit might have been a sensible precaution. However, this pattern negates many of the benefits of the cloud.

Let's step back and consider the (considerable) advantages of the cloud. The cloud is a cost-effective alternative to hosting applications on premises, mostly because of its elasticity. Deployments can be dynamically scaled up or down in response to demand, and the business pays for only the computational resources that are being used. The most exciting aspect of cloud computing, though, is the speed that it enables. Deploying to the cloud is so easy (at a technical level) that an application can be updated hundreds of times a day. This allows a business to respond to new opportunities at a lightning

pace. Innovations can be tested in the field instead of on paper, and defects can be fixed within minutes.

The engine that drives the cloud's speed is the continuous integration and continuous deployment system. This is usually shortened to *CI/CD*, and it's easy to forget that the *C* stands for *continuous*. Continuous isn't something you can buy; it's something you *do*. Sharing work continuously is challenging. A team will need a strong culture of automated testing and the discipline to start all feature development with the invisible parts, rather than the more exciting visible parts. Whereas visible parts do need to be coded before dependencies are fully ready, they need to be made invisible with feature flags. How continuous is continuous enough? For me, doing commits every 10 to 20 minutes and pushing every few commits is a good target. A team that integrates developer work less than once a day can't be considered to be practicing continuous integration, no matter how many fancy pipelines it has in place.

If integration is hard, continuous deployment is even harder. Achieving a high release cadence requires technical skill and a hospitable culture. If failure is harshly punished, a team will feel compelled to invest in lots of testing and validation before doing a release. That makes releases too expensive to do often. Even in teams with a cultural safety net, releasing regularly requires nearly 100% automation and sophisticated rollback or roll-forward mechanisms.

The Importance of Keeping Working Systems Working

Jan Urbański

Principal Software Engineer at New Relic

There's a classic phrase in the car enthusiast world: "While I'm there." Let's say you want to change your engine. You go through a lot of effort to disconnect all the externals and extract it from the chassis, and when faced with an empty engine bay, you suddenly say, "While I'm there, why not upgrade the radiator as well? And the turbocharger?" You end up with a project that takes twice as long, costs three times as much, and is leaking fluid from that ill-fitting radiator that looked so great in the catalog.

A cloud migration project is a cross-cutting undertaking, often involving touching lots of components that have previously been running smoothly for years. Everything is different in the cloud, even if vendors try their best to provide an illusion of seamlessness. Migrating means challenging all the assumptions your software makes about resource availability, network latencies, and performance characteristics of the underlying hardware. It's already a tough problem, and any extra changes you introduce to the system carry additional risk of derailing the entire effort.

The code you're migrating might be cumbersome to work with. It might be too tightly coupled, and it might be hard to configure. But it has one important characteristic: it's working. It is providing business value and has been written and maintained for a certain purpose. There are many reasons to decide on a cloud migration. These include reducing overall cost, making operations more flexible, dealing with seasonal load, or reducing the upfront investment required for growth. Regardless of your reasons, if the existing system has the desirable quality of not being broken, you should resist the urge to tinker with it as you move its components into the cloud.

Obviously, there are no absolutes. Some systems are architected in a way that makes it impossible to meet the goals of your migration project. However, to increase the chances of a successful migration, you need to take a long, hard look at any change you're introducing. Is making the change strictly required to meet your migration goals, or are you doing it "while you're there"?

Twenty years ago, Joel Spolsky wrote the classic blog post "Things You Should Never Do, Part I." He talks about how rewriting code from scratch is the single biggest mistake a developer can make. I believe the same principle applies to rewriting services from scratch. Most of the time (remember, no absolutes!), doing a lift-and-shift will lead to a better outcome than trying to take advantage of the move to refactor or rewrite existing code. Sometimes the initial migration won't even fully bring about the cost or operational benefits you are looking for. Instead, it will be a stepping-stone to further rearchitecting to better leverage the cloud environment. This can be done more calmly if the system is already up and running in the cloud.

It's easy to give in to the siren call of redoing everything, when you're already changing so many things. But a cloud migration is a big job, so if possible, avoid making it any harder on yourself. The easiest way to end up with a working system is to *keep it working*, not to *make it work*.

Effectively Navigating Organizational Politics

Joshua Zimmerman

Senior Platform Operations Engineer at Sportsengine Inc.

Where there are people, there are politics. Despite the negative views that many engineers have of office politics, they play an important role in our organizations. Put simply, politics is the way humans make collective decisions. Pain from organizational politics is a result of broken processes. The ever-increasing complexity of building and running software in the cloud makes understanding and navigating these processes an essential skill for shaping our socio-technical systems.

Before engaging in organizational politics, you need to identify the political structures in your organization. When you see that a decision has been made, ask yourself the following questions: Who was (or was not) involved in making the decision? Who does this decision impact? What did that process look like? Are the decision makers perceived as having the authority to make this decision? Who is accountable for the impact of this decision? The patterns that emerge from the answers to these questions provide the beginnings of a map of political structures. You'll even find patterns across organizations. For example, flatter organizational structures often produce more complex political structures as authority is obscured by the lack of structural definition, while members of more hierarchical organizations may have less agency to change things outside their teams.

Once you understand how decisions are made, you can influence the decision-making processes. All decisions are contextual. They're made by people with specific knowledge and under specific constraints over a given period of time. Healthy collective decision making ensures that the people who have the most context and the people most impacted are able to influence a decision. Let's look at a few ways we can do this.

Delegation

We are used to hearing about delegating work, but we can also delegate a *decision*. This is useful in a cloud environment in which one team may have multiple options to implement something that will impact another team. For example, an operations engineer instrumenting a legacy application for the first time can delegate the decision of what library to use to the development team. This decision will impact the developers throughout the application's life cycle, even though the operations team may implement it initially. To be successful, delegation requires the following:

Options
: Give people options that you are comfortable with them choosing.

Impact
: People need to actually care about the decision you are delegating.

Context
: You cannot delegate to someone who lacks the context to make a decision.

Committees

Committees can be overly bureaucratic and slow, and may not have the power to implement their decisions. But we keep trying to reimplement them (even when we rename them to something like "guilds"), because committees are a useful tool when cross-organizational context is needed for decision making or when cross-organizational buy-in is needed for a decision to be successfully implemented. To be successful, a committee needs the following:

Authority
: A committee without authority delegated to it cannot make the organization move on its decisions.

Structure
: Committee members need to have a shared understanding of how they make decisions.

Representation
: Anyone impacted by a decision needs to be involved with making it. By the same token, committee members need to represent the committee and its decisions back to their own teams.

Soft Decisions

You can make a preliminary decision in a small group and then request feedback from a larger body of stakeholders. This allows you to modify a decision based on the needs of those it will impact. Requests for comments (RFCs) are good examples of this. This approach requires you to provide as much context around your decision-making process as possible to alleviate potential concerns. Be willing to accept feedback; people will stop giving you feedback if they feel you are not listening.

Collective decision making is hard. When done well, it requires us to balance the needs of our organizations and all of the humans who are a part of them. Cloud engineers need to understand these human aspects when making decisions to reduce political friction from the decisions they make every day.

The Cloud Is Not About the Cloud

Ken Corless

Executive VP for Technology, Offerings & Partners, DXC Technology

When our clients say they want the cloud, I don't believe they really want the cloud.[1] What they really want is a whole new way of delivering technology and IT to power their business. In other words, their technology organizations want APIs and DevOps and Agile and loosely coupled systems. They want continuous deployment and autonomics and machine learning and mobile. They want to be Netflix or Instagram or Uber. They want to have two junior employees talk about an idea at lunch and work through the nights and weekends to deploy a new capability that delivers a 5% bump in sales by Monday.

Most of the CIOs of these companies understand that they must fundamentally change the way they run their IT function if they are to fulfill the aspiration that "every company is a technology company." However, while the cloud can be a catalyst toward breaking the inertia of the old way of running IT, the journey is strewn with challenges.

Many Fortune 500 companies began this journey to the cloud over 10 years ago. They called it *private cloud* because they were concerned about security, regulations, costs, retooling, and a whole host of other things, so they jumped on the private cloud bandwagon. The least successful of these companies simply slapped the private cloud label on what they were already doing. Others built out new datacenters, perhaps on hyper-converged infrastructure, and virtualized their servers, storage, and networks. Today, however, few are happy with their private cloud outcomes. Many of the more adept companies achieved some real benefits in hardware utilization and costs but others did less well.

1 A version of this article was originally published at LinkedIn.

Why did these companies not achieve their aspirations? Well, as in previous waves of technology change, they failed to change enough. Rarely will companies succeed with transformational objectives by moving only a single piece of the complex machine that is IT. Moving off the mainframe didn't do it. Nor did leveraging a cheaper global labor pool. Taking your old "pile of stuff" and pulling one lever of change simply results in a slightly updated pile of stuff.

So what should a company do that is seeking to look, act, and, most importantly, deliver outcomes significantly better? These companies need to realize that it is time to undertake a true, full IT transformation.

Companies have used the term *IT transformation* liberally in the past decade or so, even when they have only outsourced or labor-arbitraged their old pile of stuff. To fully transform IT, companies need to holistically look at their IT function without fear of breaking some glass.

You're doing DevOps? How many fewer infrastructure support people do you have?

Love SaaS? Why is your new SaaS team the same size as the old team that supported the legacy app?

You've bought into the whole API thing? Do any external parties contribute to your revenue by utilizing your APIs? Do your frontend developers access the backend of their application through the same APIs that other teams are supposed to use?

Stop. Stop now. Rethink your organization, including business interlock. (While you're there, stop talking about the business and IT as two separate entities.) Reengineer your IT business processes (the business of IT). You must take all of these wonderful, thought-out, intelligent new things that smart technology companies are doing and do them together.

Be bold. Be brave. Risk some failure. Disrupt yourself. Oh, yeah—and move your stuff to the cloud!

The Cloud Is Bigger than IT: Enterprise-Wide Training Strategies

Mike Kavis

Managing Director of Technology/Cloud Practice, Deloitte Consulting

Many people think that cloud computing is an IT project, but its arc of influence and organizational impact reach far beyond the IT department to every part of the company.[1] It is critical to be aware of these impacts. If you aren't, you risk running into roadblocks that can cause major setbacks in cloud adoption.

Companies often overlook recruiting, retaining, and training staff both inside and outside IT when planning and budgeting for a large cloud initiative. The enterprise leaders driving the cloud transformation need to work closely with their human capital team to design and build the right talent strategy. Here are some things to consider.

Gartner journalist Meghan Rimol estimates that "insufficient cloud IaaS skills will delay half of enterprise IT organizations' migration to the cloud by two years or more." She notes that many cloud migration strategies are geared toward lift-and-shift as opposed to modernization or refactoring, an approach that results in less development of cloud native skills among staff.

Include someone from human capital on your cloud leadership team from day one. Make them part of the journey. They can help build out training, retainment, and recruiting plans iteratively, and participating will keep them in sync with the progress of the overall cloud transformation. This is important because when the hiring requirements start to scale up, the recruiting team needs to be ready.

1 Excerpt from *Accelerating Cloud Adoption: Optimizing the Enterprise for Speed and Agility* (O'Reilly, 2021).

Your human capital team should be very good at building recruiting strategies, but needs your help. Provide team members with training on cloud computing so they know, at minimum, the value proposition of the cloud for the company and have a sense of what all the terms mean. Second, they'll need your help defining all the roles and job descriptions required for the cloud. Third, they'll need your ideas on where to find talent. Are there conferences or local events that they should be attending? What specific sets of hard and soft skills are you looking for? How much work can be done remotely, and how much will have to be on site?

One thing that companies leading successful cloud adoption initiatives have in common is that they make a significant financial commitment to training their employees inside and outside IT. The cloud service providers offer training programs that teach employees how to build and run workloads in the cloud, but what they don't teach is how cloud is done within *your* organization. Every organization has its own processes, policies, and controls, and its own cloud strategy, all of which play an important role in cloud technology decisions.

The most successful companies create enterprise-wide internal training programs, or *tech colleges*, that combine online training from vendors with significant homegrown training content, fully staffed with instructors and content creators. They offer these training programs not just to engineering but to the entire company.

Internal training programs should have two major focus areas. The first is upskilling employees for the cloud; the second is constantly communicating the company's cloud vision, strategy, and cultural messaging. These are critical. Organizational change guru John Kotter calls this the *WIIFM*: "What's in it for me?" If people don't know their WIIFM, it can be challenging to get them to buy into the vision and thus to the organizational changes needed.

A big part of cloud adoption is communicating the overall vision. Why should each employee care about the cloud transition? The more cloud-savvy your non-IT people are, the more effective they'll be when working with their IT counterparts.

Systems Thinking and the Support Pager

Theresa Neate

QA Practice Lead at Slalom Build Australia

A developer and I once discussed quite "enthusiastically" the consequences of the decisions we were making as a delivery team. The API we developed was to specification. It was beautifully documented. We had tested it thoroughly and were proud of it. By all appearances, our job was done. But we still didn't know how the downstream consumers of the API would receive it.

When I asked about that, the developer responded, "We don't care about that." He felt that it was up to them to adjust their consuming service based on what we gave them, seeing as they had been clearly told what we were delivering to them and we had "done it right." What ensued was our impassioned "discussion" about why that *should be* our business. We agreed to disagree.

There Are Always Consequences

Everything we do has consequences, good or bad. That is because everything—while it sometimes may appear independent—is always *interdependent*. Interdependence is fundamental to systems. Everything is a system. Our lives are systems. Everything is also a component of a bigger system.

In that scenario, as it turns out, we had not known about one of the mobile application team's requirements, and that team's whole application failed upon receipt of our API. The result was several weeks of API rewriting and retesting. (This, of course, is *waste*, a topic for an essay on Lean crucial to successful DevOps—but more about that on another day.)

The consequences of poor software design, development, and delivery are always felt, whether it be by the operations folks, or the customers, or the shareholders, or whoever. Likewise, the consequences of teams that work together, maybe even on the same floor, but do not consider the effects of their actions on others will be felt.

Systems Thinking in Teams

Teams do not work in isolation; they are a part of a system. Nowadays, we use terms like *DevOps* and *Agile* to describe this culture. (Note that *culture* is how we deal with work, not how many beanbags we have.)

While tools help us achieve these aims, it is systems thinking and the culture that actually get us there. When we enable and educate our teams to be part of a system, team members consider not only the individual components of this system but also the system as a whole—and the consequences of their actions.

Systems Thinking in Application Support

The person who wears the support pager is the person who has the best view of the system's bad behavior and the system's supportability. This person knows all too well the consequences of the actions taken by those earlier in the life cycle.

Traditionally, this person is operations personnel. However, in mature DevOps practices, other personnel in the team carry the pager too. At my current employer, all developers and some QAs do too. We do this because we espouse ownership (and considering consequences) of the work our teams do.

In my case, I am not a systems engineer, and when I am personally unable to resolve the pager alerts, I seek the assistance of someone who can. They are usually very willing to do so because they benefit *directly* from educating the rest of us.

It All Dovetails

So much can be said and written about the dovetailing of systems thinking, Lean, Agile, and DevOps. But for now, I will leave you with this summary and leave the door open to further discussions.

Supporting a system and empowering team members to support their system aids understanding of what causes issues, provides insight into regularity of recurring issues, and improves empathy and ownership—and quite rightfully sets everyone on the path of thinking in terms of consequences, and thus systems thinking.

Curating a DevOps Culture and Experience

Tiffany Jachja

Tech Evangelist at Harness

DevOps came from the idea that by working together, developers and operations teams could drive the delivery of business value through software. I came to understand DevOps as a combination of people, process, and technology.

Admittedly, it took me quite some time to learn how these three areas played into each other. So I want to share three lessons about adopting DevOps practices and culture for developers, product owners, and stakeholders within organizations.

Define Your Target Outcomes

I've come to realize that things can go very wrong in any endeavors where the outcomes are not clearly defined and prioritized. Output and artifacts are significant; we produce them in design thinking sessions, feature building, and discovery sessions. However, these outputs and artifacts can quickly depreciate when teams misunderstand the goals and purpose.

The critical point here is not to confuse outcomes with outputs. You shouldn't run a design sprint to produce a backlog for your developers to develop off of; this is an example of an *output*. An example of an *outcome* is having your critical stakeholders, including your developers, understand the problem space to begin developing the solution. If you're running a session or event and have artifacts and checkboxes to produce, that's great; it ensures someone has a role and responsibilities to fill during that time. These responsibilities should be a separate agenda handled by that role.

When a team aligns with the same outcome, we get better value and results. *Accelerate* by Nicole Forsgren et al. (IT Revolution Press, 2018) is an excellent book that talks about this within organizations adopting DevOps. Defining target outcomes ensures that processes involving tech and people are aligned with your business needs.

Safe Environments

Safe environments are essential to maintaining the health of your organization. Employees can bring their best only when they feel empowered and motivated. The key takeaway here is to provide a safe environment for everyone to contribute.

If you are maintaining a healthy team, excellent—you are probably meeting your target outcomes, driving value across the organization or group regularly, and have lessons learned. Great feedback often causes a desire to further improve or experiment with current processes.

If you are struggling with delivering value and meeting your target outcomes, a few practices can help enable learning from your team, like retrospectives. *Dare to Lead* by Brené Brown (Random House, 2018) is a personal favorite of mine for grappling with conflicts and complexity within an organization.

Regardless, we cannot predict chaos, so within execution cycles it's often necessary to pivot. Changing current outcomes and processes across your organization can be difficult, especially concerning team morale and team dynamics. Blame can also be associated with dissatisfaction tied to unmet deliverables.

Some indicators of lowered team morale include feelings of frustration that things aren't working, pushback, and an increase in questions. Here are some tips if you are experiencing mid-sprint or mid-execution pivots that are hurting your team:

1. Ensure that the change aligns with a defined target outcome.
2. Constrain the change by time or ensure that the shift is time-bound (especially if you are making changes mid-execution).
3. Explain the change to your team in the context of points 1 and 2.

Maintaining a safe environment will help grow a culture in which individuals gain autonomy, allowing value to scale.

Architect Your Technology

Organizations see the need to adapt and innovate to stay competitive and current in their markets. With the increasing adoption and emergence of technology—there are 1,200 Cloud Native Computing Foundation (CNCF)

projects now, wow!—it's crucial that team members across the organization understand the technology landscape of your company.

Ensure that documentation is current and available regarding the architecture of your application. If you do not have any architecture diagrams, take the time to work with your team to produce, at a minimum, a big-picture diagram that spans your environments. Present this to your organization and ensure that everyone understands how all the tools and frameworks interact. With reference diagrams, your developers can understand the software application and point out any missing details concerning current work in progress.

Getting the big picture also allows you to better determine which processes and features will require or benefit from the adoption of technologies. Understanding how your tools and technologies benefit your software development life cycle accelerates your software delivery.

PART XII

Personal and Professional Development

Read the Documentation—Then Reread It

Jennine Townsend

System Administrator

Moving to the cloud can really reduce the number of infrastructure vendors you'll need to keep an eye on. But you can't slack off on reading documentation, since understanding your cloud vendor is as important as understanding all those others used to be, combined, and if you don't know about a feature, it might as well not exist.

But even if you read the documentation, another challenge remains: cloud providers are constantly working to improve their offerings. In many cases, the improvements will appear as new services or splashy new features, and you'll find out about them in the normal course of keeping up. Follow all the vendor's tech blogs! But often the improvements are to a service you aren't now using, or you just don't happen to notice the announcement, or maybe it never gets officially announced. How can you know to use a feature you don't know exists? Even the infrastructure you're already using may be changing right this minute! The documentation you read last month might already be out of date.

It's helpful to have a strategy. Brain upgrades require as much intentional planning as any others.

DevOps practitioners already ought to have a good understanding of infrastructure costs and the contribution of various technologies and projects to the overall costs. Always keep up with the documentation around the infrastructure that's costing the most, or that is bottlenecking the highest-priority projects. There's a good chance that this isn't the trendy new tech, but causing random sudden cost savings can be pretty rewarding too.

In addition, the cloud by its more on-demand nature adds emphasis to calendar considerations: for example, reservations or licenses may apply to some resources, so with advance planning, you can take advantage of expirations to switch to cheaper and better choices. That advance planning requires understanding updated business needs and how the related

offerings and pricing have changed since the reservation was purchased, so reviewing upcoming reservation expirations is an excellent way to choose documentation to focus on.

What is causing too much toil and trouble? A big advantage of the cloud is that you don't have to live out the lifetime of a bad technology, but wholesale replacement can be a project too large or risky to ever complete. If some part of the stack isn't working well, a simple tweak may help. Even, and maybe especially, if you don't have specific words for your dissatisfaction, it can really pay off to reread the documentation; you may find that you're suffering with something that already has a fix available! As with sudden cost decreases, a sudden improvement in usability or reliability or performance can be really welcome, especially if it carries little or no burden of code changes.

Then there are the pieces of infrastructure that just feel old. Here's where you get to play with the shiny new thing—but not until after you've reread the docs for the old thing! At a minimum, it's best to have a solid and updated understanding of what you're replacing. This is far easier if your stack is designed to be as modular as possible, so that you can swap out the queue layer, say, without much disruption.

The cloud brings fundamentally different trade-offs to many of the careful balancing acts that come with developing and operating an infrastructure. The flexibility of ephemeral infrastructure gives us the ability to rapidly improve the platforms that support our projects, but we can take advantage only of flexibility that we know we have. Reading—and then rereading—the documentation is important for making wise decisions about where to most effectively apply time and effort, but keeping up is a big project in itself. Defining a strategy for *how* to keep up can make doing so more manageable and useful.

Read the documentation. Then reread it!

Stay Curious

Laziz Turakulov

Digital Business Analyst at BAT

As a cloud engineer, you should know a lot. And the more you learn, the more you realize that the learning has no ending. It becomes a lifelong journey, with new technologies, platforms, and solutions released almost daily!

That's why it's important to make the learning process as rewarding as possible, so that you can enjoy every byte of information building another synapse in your brain. And the secret ingredient for this is *curiosity*.

Look at babies for inspiration. They are full of natural curiosity, trying to touch and taste almost anything they can reach. Unfortunately, as adults, we get busy with various priorities at work and home, and lose our natural sparks of curiosity.

However, they are not completely lost! As a cloud engineer, you can do the following exercises to reignite that joy of trying something new and boost your career in the field:

Switch off the bias.
: You may be a hardcore (and hardcoded!) fan of a certain cloud platform, hate or love open source, and believe that your favorite programming language beats them all. It may be true. Or maybe not. So don't limit yourself to only what you know or prefer, but instead allocate regular slots in your busy diary to explore competitor products. You may not like them at all, but at least at the next meeting you'll be able to explain (without getting too emotional) the differences between the options and how the client would benefit from your recommendation.

Learn about business processes.
: Gone is the time when you could lock yourself down in the server room. As a cloud engineer, you are on the front line and should speak "business" language, understanding what people in the marketing or operations departments do, what data business analysts dream of, and what keeps your security team awake at night. The more you learn about

those processes, the better you can bridge client requirements with the relevant cloud offerings.

Not everything is about technology.
: Try to find a new hobby: music, art, foreign language, or sport. The link may not be obvious, but it may help you look at some of your challenges from a different angle, find common topics for small talk with colleagues, establish new networks, and boost creativity in general.

Try something that looks impossible, stop to refocus and revise, and then try it again.
: Babies fall, but they get up again and again, eventually learning how to walk. In your job, you may try a slightly different approach with every new attempt. As one famous and wise man once said, "doing the same thing over and over again, but expecting different results" is the definition of something else. He probably was never a baby himself!

Teach others.
: When you need to explain something to others who are not that familiar with the subject, think about how to make the information simple and digestible. No jargon, no condescension—you were not born a cloud engineer yourself! Help others to understand it, and you will learn more about the subject too (possibly making new friends along the way).

But most of all, enjoy what you do!

Empathy as Code

Nirmal Mehta

Chief Technologist at Booz Allen Hamilton

One of the many reasons for the rise in popularity of cloud computing and DevOps is in response to the established ineffective tension between development and operations teams seen across most of the enterprise IT landscape. The often dysfunctional pattern of a throw-it-over-the-fence approach to responsibility created an environment in which any change was often met with fear and resistance.

I hope to provide some context for all the technologies, tools, architectural patterns, techniques, and other material you are absorbing in this book. I also encourage you to utilize infrastructure as code as common ground between developers and operators.

Empathy as Code

Many of the tools and techniques in this book are centered around writing code that defines and implements various aspects of the infrastructure you are working on; this is commonly known as *infrastructure as code* (IaC). Whether they're Terraform modules, Dockerfiles, Ansible playbooks, or Kubernetes YAML/Helm charts, these pieces of code codify *decisions* that are being made by cloud engineers to dictate what the infrastructure should be to run the developers' applications.

In addition to the benefits of automation, repeatability, and security that these IaC elements provide you as a cloud engineer, they can also be used to bridge the gap between operations teams and application developers. IaC can be used to shift the throw-it-over-the-fence process into a *shared responsibility* process in which joint decisions on what the infrastructure should look like are codified.

IaC then turns into a form of *empathy as code*, through which groups of folks such as developers, security engineers, and cloud engineers can find common ground, understand incentives and goals, test new designs, and share responsibility for the system as a whole. As a cloud engineer evaluating, designing, and implementing new technologies, I encourage you to take

a step back and find opportunities to use these technologies to break down the organizational walls and foster understanding.

A Sampling of Decision-Making Techniques

Great! You're now on board for DevOps, infrastructure/empathy as code, and shared responsibility—but how do you go about making architecture and technology decisions with a group?

Here is a sampling of decision-making and forecasting techniques to get you started:

Design thinking
: First described by Nobel Laureate Herbert Simon, this popular innovation process takes a user-centric approach: empathy for stakeholders/users, gathering requirements by understanding the users' needs and problems and by utilizing your unique insights, out-of-the-box ideation, prototyping the solution, and iterative testing with users. Design thinking is useful at the beginning of a new project or when collaborating with diverse groups on a solution to a larger problem.

Divergence/convergence
: A lightweight framework for group brainstorming and "norming" around a solution, this framework has two phases. In the *divergence* phase, any solution can be proposed, and the group is encouraged to constructively support all ideas. Then *convergence* begins: technical leadership drives consensus on a solution from the first-phase ideas.

Strong opinions loosely held
: In this process developed by Paul Saffo, you let your intuition create a *strong opinion* about a solution and then use self-awareness to gather evidence to prove yourself wrong (a form of "self-steelmanning"). This technique tends to work only if everyone is working in good faith.

Superforecasting
: Developed by Philip Tetlock in *Superforecasting* (Crown, 2015), this technique focuses on two things: phrasing opinions in a clear future-verifiable way, and stating the probability that your opinion is correct. These two steps can help open discussions between differing groups and lead to consensus.

From Zero to Cloud Engineer in Less Than a Year

Rachel Sweeney

DevOps Engineer at the Pew Research Center

A year and a half before I wrote this article, I was an executive assistant with zero years of IT experience and a degree in Chinese history. Despite that promising pedigree, I managed to get offers in four out of five of the jobs I interviewed for, and I now have the pleasure of working as a DevOps engineer, with tools and technology that have me excited to go to work every day.

So, what's the secret?

I achieved my dream by making the most of the resources that are readily available to all of us with an internet connection. I joined several Slack channels and Discord groups to talk to experts and pick their brains, I picked up books and watched videos on YouTube and Linux Academy to develop the skill sets I needed, and I went to meetups both in person and virtually to learn more and network with locals to see what their day-to-day work looked like.

When I began my journey, I didn't really know what I wanted to do, so I found a Linux Slack channel and said, "Hey everyone, I want to be a Linux system administrator—where do I start?" I was amazed at the number of responses I received. As I asked more and more questions, a path started to unfold, with a number of skills I needed to get me to where I wanted to be. Several months later, I found myself repeating these same questions. I quickly learned I wanted to be a cloud engineer, and shortly thereafter I accepted my first job as an assistant DevOps engineer. So many experts are available to us through Slack and Discord who love sharing their depth of knowledge; use this to your advantage! Ask them questions.

While talking with some of these experts, I found that the next thing I needed to do since I was starting from zero was to obtain a few certifications. Although there's plenty of debate on whether certifications will be really helpful for your next job or promotion, the knowledge gained while studying for them is invaluable. I set aggressive timelines for myself, allowing three months to study the material and attempt the exam. This was an extremely motivating factor to keep studying and push myself to the next level. In most of the interviews I've had, when I was asked how to solve a particular problem, I was able to draw directly from the material I'd learned to answer the question and secure a job offer. Getting a certain score on a certification exam isn't nearly as important as knowing the material. Even if you don't want to or can't afford to get a certification, just looking at the study material is worthwhile. Quite often, the cloud provider or company offering the certification is preparing a silver platter piled high with all of the important things they feel you should know.

The last resource that I can't recommend enough is attending local or virtual meetups in your area. Cloud engineering often involves a particular problem that needs a solution, given the tools we have available. Meetups usually revolve around problem solving, whether it's someone sharing a challenge and how they solved it, or learning about a tool and the challenges it aims to solve. Meetups are also especially useful if you're job seeking. The majority of my job interviews arose from word-of-mouth connections and talking with others at these meetups. A good icebreaker I would often use at meetups was asking people, "What's your favorite DevOps tool and why?" Don't forget to ask open-ended follow-up questions!

This book has presented an amazing array of skills, ideas, and tools that can help you on your journey, whether you're starting from scratch or are established in your career. If you apply these resources to whatever you're learning, I guarantee that you will end up with a deeper and better understanding of your topic.

Contributors

Adarsh Shah

Adarsh Shah is an engineering leader, coach, public speaker, hands-on architect, and change agent. He is also an organizer for the devopsdays NYC conference and DevOps NYC meetup. Adarsh has a keen interest in building systems that add business value. He is an independent consultant passionate about helping clients with software architecture/development, leadership, and DevOps and cloud needs by looking at both technical and nontechnical aspects. These days, he is excited about working with machine learning and cloud native technologies. Find out more about Adarsh at *shahadarsh.com* and on Twitter at *@shahadarsh*.

> *Principles, Patterns, and Practices for Effective Infrastructure as Code*, page 180

Alessandro Diaferia

Alessandro Diaferia is a software products technologist and a senior software engineer and lead engineer at Utmost, in Dublin, Ireland. He's always keen to understand how new technologies enable teams to be more effective at delivering value. Alessandro favors approaches that invest in metrics and data-driven decision making, and teams that don't become enamored with their assumptions, but rather are agile and ready to pivot as the conditions of the environment change around them. He believes cloud technologies to be the key that helps organizations experiment and quickly understand the best value for their users and believes in adopting them to reach the highest standards of agility. Alessandro spends part of his free time reading about technologies and practices and blogging about his experiences. He cohosts a podcast for Italian-speaking software engineers called SpaghettiCode. You can find him on Twitter at *@alediaferia*.

> *The Cloud Doesn't Care if It Works on Your Machine*, page 138

Alex Nauda

Alex Nauda is CTO at Nobl9 and helps organizations improve the reliability and performance of their cloud native applications. He started his career in the performance management of data warehousing in the days of magnetic storage and backplanes. Since the days of the web, he has focused on product development in media and the public cloud. Alex lives in Boston, where he grows vegetables under LEDs and teaches juggling at a nonprofit community circus school. You can find Alex on Twitter at *@Alexnauda*.

The Basics of Service-Level Objectives, page 107

Annie Hedgpeth

Annie Hedgpeth is a senior cloud automation engineer at 10th Magnitude, where she focuses on accelerating Azure cloud adoption through automation and DevOps. Through configuration management, provisioning with infrastructure as code, integration testing and compliance automation through InSpec, and CI/CD, Annie's aim is always to make the right thing to do the easy thing to do. Simplifying CI/CD and creating a seamless process is at the heart of what she does. You can find out more about what Annie does on her blog at *anniehedgie.com*, and you can follow her on Twitter at *@anniehedgie*.

Red, Green, Refactor for Infrastructure, page 183

Asha Kalburgi

Asha Kalburgi is a database architect with an interest in DevOps and CI/CD, especially for databases. She has over 16 years of experience in database technology and recently became fascinated with all things cloud. Asha is certified as an AWS Solution Architect. Currently, she is working on migration of on-premises Oracle databases into the AWS cloud, especially Aurora PostgreSQL RDBMSs. Asha has a passion for DevOps and has implemented the CI/CD pipeline for database code and worked with CI/CD tools like Jenkins, Git, and Liquibase. She believes "change is the only constant" in today's technology fabric.

So You Want to Migrate Oracle Database into AWS Cloud?, page 193

Banjo Obayomi

Banjo Obayomi is a senior research engineer at Two Six Labs, where he develops platform solutions for productizing various research-based projects. Banjo is passionate about operationalizing data, aka DataOps, and has started a podcast and meetup around data. Banjo received his BS in computer science from the University of Maryland, College Park, in 2011 and his MS in computer science from Loyola University in Maryland in 2015.

DataOps: DevOps for Data Management, page 196

Brendan O'Leary

Brendan O'Leary is a senior developer evangelist at GitLab, the first single application for the DevSecOps life cycle. He has a passion for software development and iterating on processes just as quickly as we iterate on code. Working with customers to deliver value is what drives his passion for DevOps and smooth CI/CD implementation. Brendan has worked with a wide range of customers—from the nation's top healthcare institutions to environmental services companies to the US Department of Defense. Outside of work, you'll find Brendan with one to four kids hanging off him at any given time, or occasionally finding a moment alone to build something in his workshop. You can reach Brendan on Twitter at *@olearycrew*.

Three Keys to Making the Right Multicloud Decisions, page 6

Brian Singer

Brian Singer is a product-focused entrepreneur with a passion for enterprise software, cloud computing, and reliability engineering. He is cofounder and chief product officer of Nobl9, a Battery Ventures–backed company building a platform to optimize software reliability. His previous company, Orbitera, was acquired by Google; he adapted the SaaS product to follow Google's best practices for production and reliability. Prior to Orbitera, Brian worked for BMC and Novell. Brian holds a BS in computer engineering from Brown University and an MBA from MIT. He resides in the Boston area, where he is perfecting his golf swing.

The Basics of Service-Level Objectives, page 107

Brittany Woods

Brittany Woods is an automator of things who is based in central Missouri. During her career in technical roles in both the financial and automotive sectors, Brittany has been a major advocate for utilizing DevOps and automation to enhance velocity, increase innovation, and drive business value. She enjoys work that spans not only across DevOps and automation spaces but also into systems architecture and cloud solutions and adoption. Additionally, Brittany enjoys teaching others, sharing knowledge through conference speaking, and leading teams. She can be found on Twitter (*@bnwoods2008*) or on her website (*brittanynwoods.com*).

Silos by Any Other Name, page 209

Chris Hickman

Chris Hickman is an entrepreneur, technical leader, and accomplished developer. He has founded two companies, one with $24 million in venture capital, the other bootstrapped with an SBA loan. Now at Kelsus, he leads teams building world-class software in the cloud using AWS, Docker, and the latest, most productive technology stacks. Chris also cohosts *Mobycast*, a weekly podcast that dives deep on topics related to building cloud native software. When not designing distributed systems or deploying code to the cloud, Chris enjoys spending time with his family, cycling, and endlessly throwing a tennis ball to his black Labrador, Gus.

The Future of Containers: What's Next?, page 20

Chris Proto

Chris Proto is a long-time technologist with a diverse technical skill set in security, application development, and operations. After graduating as a computer engineer from Villanova University he moved to Denver, where he worked as a developer for companies of all sizes until joining the engineering team at Craftsy, a successful consumer media startup. After supporting years of scaling, continuous growth, and an eventual acquisition by NBCUniversal, Chris left his position as head of DevOps to follow his passion and start his own company focused on promoting cloud engineering best practices. Currently, Chris resides with his wife in Charlottesville, VA,

where he owns and operates DevOps Gorilla, a professional services provider for companies running cloud native workloads. Follow Chris on Twitter (*@cproto*) or online at *https://www.devopsgorilla.com*.

KISS It, page 140

Chris Short

Chris Short has been a proponent of open source solutions throughout his over two decades in various IT disciplines including systems, security, networks, and DevOps engineering and advocacy across the public and private sectors. He currently works at Red Hat. Chris is a disabled US Air Force veteran living with his wife and son in Greater Metro Detroit. He writes about DevOps and other topics at *chrisshort.net* and runs the DevOps, cloud native, and open source–focused newsletter DevOps'ish.

Security at Cloud Native Speed, page 69

Dan Moore

Dan Moore has over 20 years of experience building software. He's been an employee, a contractor, an AWS course instructor, an author, a community member, a meetup organizer, an engineering manager, and a CTO. Dan currently leads developer advocacy at FusionAuth, a company building software to handle authentication, authorization, and user management for any app. When not in front of the keyboard, you can find him in the wilderness or the garden. Dan resides in Boulder, CO. You can find him on Twitter at *@mooreds*.

Use Managed Services—Please, page 8
When in Doubt, Test It Out, page 94

Dave Stanke

Dave Stanke is a developer advocate for Google Cloud Platform, aligned to the DevOps community. He loves talking with practitioners: listening to stories, telling stories, and sharing a healthy cry. Before moving to Google, he was the CTO at OvationTix/TheaterMania, a tech startup in the performing arts industry, where he specialized in feeding memory to Java servers. He chose on purpose to live in New Jersey, where he enjoys baking, indie rock, and fatherhood.

People Will Expect Things—Help Them Expect Right, page 59

David Murray

Proudly Irish with 20 years of experience in customer-facing technology roles, David Murray knows the tech may change but the customers never do! David is a networking engineer at heart who is working on cloud solutions at the moment. He loves nothing more than trying to tackle the business or technical problems that are so big that most people shy away from them. In his spare time…who are we kidding, he has two young boys and zero spare time. His side hustle is that he is the head coach for the LA Cougars Gaelic Football team.

Even in the Cloud, the Network Is the Foundation, page 202

Dawn Parzych

Dawn Parzych (*@dparzych*) is a developer advocate at LaunchDarkly, where she uses her storytelling prowess to write and speak about the intersection of technology and psychology. She enjoys helping people be more successful at work and at life. She makes technical information accessible, avoiding buzzwords and jargon whenever possible. Her articles have appeared in numerous technical publications. She serves as an organizer for Write/Speak/Code, the Seattle DevOps Meetup, and is on the organizing committee for devopsdays Seattle. In her free time, she enjoys exploring the Pacific Northwest with her family and dog.

Maintaining Service Levels with Feature Flags, page 142

Deepak Ramchandani Vensi

Deepak Ramchandani Vensi, a transformation director at Contino, specializes in working with large regulated enterprises to accelerate their journey toward digital transformation and multicloud adoption. He works with technology executives to share strategies and approaches on pivoting toward a digital-first business and operating model. Deepak is known for helping organizations adopt modern operational and engineering practices such as SRE, FinOps, GitOps, and DevSecOps; developing cloud-native products; and building sustainable in-house digital capabilities for long-term adoption across Azure, AWS, and GCP. Over the past few years, Deepak has worked with leading brands and institutions to overcome their internal barriers to change by moving toward modern ways of working, focusing relentlessly on the developer experience and adopting the public cloud, while respecting the boundaries set by the respective regulatory bodies. This includes banking and financial services, insurance, retail, business process outsourcing, energy, and utilities.

FinOps: How Cloud Finance Management Can Save Your Cloud Program from Extinction, page 165

Delali Dzirasa

Delali Dzirasa is the CEO and founder of Fearless, a full-stack digital services firm in Baltimore, MD, with a mission to create software with a soul—tools that empower communities and make a difference. In addition to shooting for the stars when it comes to ideas and living the purple cow principle, Delali has set the vision for Fearless for more than 10 years. Delali has a BS in computer engineering from the University of Maryland, Baltimore County (UMBC), and has over 15 years of experience leading Agile software teams and programs. He strives to make a difference in technology and in his surrounding community.

Cloud for Good Should Be Your Next Project, page 10

Derek Martin

Derek Martin is an accomplished Microsoft systems developer and integrator and a principal program manager for the Azure Patterns and Practices Team. Derek has 15 years of experience in developing and deploying public and private clouds and integrating line-of-business applications with modern technology. His extensive experience in cloud technologies and digital transformation helps him guide clients on their journey to the cloud. He's focused on developing and articulating Azure and general cloud best practices around governance, networking, security, and computing.

Essentials of Modern Cloud Governance, page 72
Never Take a Single Region Dependency, page 96
Networking First, page 204

Duncan Mackenzie

Duncan Mackenzie, *@Duncanma* on Twitter, is the engineering manager for several Microsoft websites including *docs.microsoft.com*, *azure.microsoft.com*, and *channel9.msdn.com*. Over the years, he has written extensively on programming topics at Microsoft and in a series of books. These days, his writing can be found on his own blog at *www.duncanmackenzie.net*, where he likes to focus on his passion for web performance. Outside of tech, he enjoys spending time with his family, traveling when they can, and exploring the Pacific Northwest when staying closer to home.

Understanding Scalability, page 23

Emily Freeman

Emily Freeman is a technologist and storyteller who helps engineering teams improve their velocity. As the author of *DevOps for Dummies* (Wiley, 2019) she believes the biggest challenges facing developers aren't technical, but human. Her mission in life is to transform technology organizations by creating a company culture in which diverse, collaborative teams can thrive. Emily is a principal cloud advocate at Microsoft and lives in Denver, CO, with her daughter.

Emmanuel Apau

Living and working out of the Shire in the District of Columbia, Emmanuel Apau, a first-generation Gambian–Ghanian American, spends his free time ruminating on the final season of *Game of Thrones* that could have been. He is the CTO at the consulting firm Mechanicode.io, cofounder of the Black Code Collective, and a recipient of DC's Technical.ly RealLIST Engineer award. Emmanuel is an AWS-certified DevOps specialist with 10 years of experience developing innovative automation solutions using DevOps and site reliability best practices for clients. He has a passion for automation and improving the feedback loop in the developer experience. Emmanuel has experience in both the public and private sectors, providing modernization services that engage Agile best practices, scalable cloud architectures, and continuous integration and deployment standards. Twitter: *@technoGrouch*.

Know Where the Secrets Are Kept and How, page 75

Eric Sorenson

Eric Sorenson has been working in systems administration since 28.8k modems were exotic luxuries. After running campus networks, large-scale production internet services, and sysadmin teams, he moved to Portland, OR, in 2012 to work at Puppet as a technical product manager for their core technology platform. Since 2018, he's been focusing on building awesome products for cloud-centric DevOps engineers.

Working Upstream, page 145

Fernando Duran

Fernando Duran, born in Cadiz, Spain, holds a BS in physics and an MS in computer science. He has acquired a broad skill set by working in various roles including researcher, sysadmin, software engineer, infosec practitioner, and team leader. He loves watches, poker, soccer, bad beer, good coffee, and reading about ancient civilizations. Currently he is the DevOps lead at Kira Systems—the world leader in automated contract review—and lives in Waterloo, Ontario, Canada, with his talented artistic wife and two squatters. Twitter: *@fduran*.

Don't SSH into Production, page 78
Test Your Infrastructure with Game Days, page 98

Geoff Hughes

Geoff Hughes is a technology leader who has delivered results at Fortune 100 companies Cisco and Wells Fargo, and across the technology, financial, and software industries. He built Cisco's first global systems administration team and was the first recipient of the Cisco IT Global Operations "Excellence in Operations" award. Geoff also led the Cisco IT team that deployed the first Cisco MDS, disrupting the SAN market dominated by Brocade and McData. As head of storage operations for Wells Fargo, he led a global team that delivered improvements in storage availability while reducing operational expense. As the director of SaaS Infrastructure Architecture and Engineering at CA Technologies, he led a global organization that scaled infrastructure to support growth from 3 products to 14, resulting in annual revenue growth from $180 million to $540 million. Currently, he is leading site reliability engineering for NetApp's Cloud Volume Services in AWS and GCP. Geoff loves spending time with his family and playing soccer.

Data Gravity: The Importance of Data Management in the Cloud, page 198

Guillaume Blaquiere

Guillaume Blaquiere has been a Google Developers Expert on the Cloud Platform since 2019 and works at Veolia as lead cloud architect. He has been a Java developer for more than 15 years, and despite various responsibilities has always kept his focus on creating, developing, discovering, and testing new solutions, especially in the cloud, using machine learning or Python and Go.

Focus on Your Team, Not on the Cost, page 211

Haishi Bai

Haishi Bai is a principal software architect at Microsoft working on innovative projects across the cloud and edge. He's been coding for over 30 years and has been working as a software professional for 23 years. He's a believer in continuous learning and open knowledge sharing. He has authored nine cloud computing books and he's been a volunteer teacher at high schools for four years, teaching programming skills. Haishi is the cocreator of open source projects such as OAM and Dapr.

Don't Think of Services, Think of Capabilities, page 25

Holly Cummins

Holly Cummins is the worldwide development practice lead for the IBM Cloud Garage. Holly delivers technology-enabled innovation to clients across a range of industries, from banking to catering to retail to NGOs. She has led projects to count fish, help a blind athlete run ultra-marathons in the desert solo, improve healthcare for the elderly, and change how city parking works. Holly is also a Java Champion, IBM Q Ambassador, and JavaOne Rock Star. Before joining the IBM Cloud Garage, she was delivery lead for the WebSphere Liberty Profile (now Open Liberty). Holly coauthored *Enterprise OSGi in Action* (Manning, 2013).

Cloud Engineering Is About Culture, Not Containers, page 213

Isuru J. Ranawaka

Isuru J. Ranawaka is a senior full-stack cloud developer for the Cyberinfrastructure Integration Research Center (CIRC) at Indiana University. He has worked on identity and access management projects for science gateways, transport layer developments for an enterprise service bus, and E2E application development in Java, Node.js, and React. Isuru has more than six years of industry experience in software development. Primarily, he is interested in cloud computing, microservices-based software development, NoSQL-based data management, CRAN-based network architectures, and communication technologies.

Identity and Access Management in Cloud Computing, page 80

Ivan Krnić

Ivan Krnić is director of engineering at CROZ, striving to create the best possible conditions for teams to move forward. His special areas of interest cover Agile software development, cloud native architectures, complex integrations, and DevOps culture. As a developer and project manager, he has seen it all, and now he hopes to dent the universe as an Agile and DevOps coach. Particularly interested in leadership and organizational change, he is helping organizations align business and tech, focus their efforts, and essentially work smarter, not harder. Being an Agile enthusiast, he is an active member of the community, periodically holding courses and giving talks on various Agile and DevOps topics. Ivan is a hopeless sucker for start-up ideas, although all of them have failed so far. He enjoys windsurfing and running. You can find him on Twitter as *@ikrnic*.

Do More with Less, page 148

Iyana Garry

Iyana Garry is a CompTIA Security+ and CCNA-certified freelance security researcher with a handful of help desk experience. She has an affinity for cultivating skills in cloud computing, coding, and security via self-taught learning as well as helping aspiring IT professionals start their careers. Her Twitter handle is *@theautom8er*.

Treat Your Cloud Environment as if It Were On Premises, page 83

J. Paul Reed

J. Paul Reed began his career in the trenches as a build/release and operations engineer. After launching a successful consulting firm, he now spends his days as a senior applied resilience engineer on Netflix's Critical Operations & Reliability Engineering (CORE) team, focusing on incident analysis, systemic risk identification and mitigation, applied resilience engineering, and human factors expressed in the streaming leader's various socio-technical systems.

REvisiting the Rs of SRE, page 103

Jake Echanove

Jake Echanove started deploying and managing SAP in cloud-like environments in early 2011. He has helped dozens of customers move their mission-critical applications to the cloud, while focusing on the transformational opportunities that the cloud provides for legacy applications. He has given many talks on this topic at events including SAP Sapphire, Dell Tech World, and VMworld. Jake has held a variety of positions in delivery, operations, and presales. Most recently, he is SVP of Solutions Architecture for an AWS global partner. Jake loves to travel, eat, root for the 49ers, and spend time with his wife and two children. Follow him on Twitter: *@jakeechanove*.

You Can Cloudify Your Monolith, page 27

Jan Urbański

Jan Urbański is a principal software engineer at New Relic, where he is primarily focused on the backend data pipeline. Previously he cofounded Ducksboard, a dashboarding SaaS company acquired by New Relic in 2014, and before that he worked with audio and video streaming. He is a PostgreSQL contributor and has also had code accepted in other open source projects, such as GStreamer, Twisted, and Apache Cassandra. His interests revolve around database internals, distributed systems, and Linux.

The Importance of Keeping Working Systems Working, page 215

Jason Katzer

Jason Katzer is the creator of CloudPro.app, which creates developer productivity tools for the cloud, and offers consulting on cloud native architectures and cloud cost savings. Previously, he served as director of software engineering at Capital One (Paribus/WikiBuy) and at Blink Health. Jason is also a serial entrepreneur and angel investor who's been involved with and started many new ventures. He's worked in several industries including healthcare, consumer tech, fitness, sales, finance, and telecom and loves to help people save both time and money—but his one real focus is on building quality software. He is a passionate teacher (Make School) as well as a lifelong learner. He devours TV shows, podcasts, and audiobooks and will dearly miss the voice of Vin Scully.

Improve Your Monitoring with Visualizations and Dashboards, page 101

Jennine Townsend

Jennine Townsend has been a system administrator and documentation enthusiast for a long, long time. As a junior sysadmin, one of her tasks was to swap in the documentation updates for the VAX/VMS operating system, which periodically arrived as stacks of updated pages that had to be swapped into the physical documentation notebooks. This was an effective means of training a junior, and seems to have cemented an awareness of the value of keeping abreast of updated documentation. Since then, she has seen the job change and even get new names; programming languages come and go, companies rise and fall, but fundamentally it's still about helping people work together to effectively use technology.

Read the Documentation—Then Reread It, page 230

Jon Moore

Jon Moore is the chief software architect at a Fortune 50 company, where he focuses on leading the company to continually sharpen delivery of its software-based products. He specializes in the "art of the possible," finding ways to coordinate working solutions for complex problems and deliver them on time (even in large enterprises). Jon is equally comfortable leading and managing teams and personally writing production-ready code, and has a passion for software engineering—continuously learning and then teaching colleagues new ways to deliver working, maintainable software with ever-higher quality and ever-shorter delivery times. His current interests include distributed systems, fault tolerance, building healthy and engaging engineering cultures, and Texas Hold'em. Jon received his PhD in computer and information science from the University of Pennsylvania and resides in West Philadelphia, although he was neither born there nor raised there and does not spend most of his days on playgrounds.

How Economies of Scale Work in the Cloud, page 168

Jonathan Buck

Jonathan Buck received his bachelor's and master's degrees from the Georgia Institute of Technology, and then made his way to the West Coast to pursue a career in technology. He is passionate about the opportunities afforded by cloud computing, and enjoys continuous learning and sharing of best practices with others.

A Cloud Computing Vocabulary, page 12

Joshua Zimmerman

Joshua Zimmerman has been in tech for the past decade, in a variety of roles and positions. Joshua is passionate about creating sustainable platforms for applications and currently does this for SportsEngine. He is prone to go on long rants about how you should respect libraries, universities, and the public sector more than you do currently. In his spare time, he tweets about his cats and helps organize DevOps things in Madison, WI.

Effectively Navigating Organizational Politics, page 217

Judy Johnson

Judy Johnson has been a software engineer for a long time, and has been at Onyx Point since 2015. She has also functioned as a systems engineer, project manager, scrum master, and CD store clerk. When not at work, Judy can be found baking yummy treats for family, friends, and coworkers; attending hockey games and rock concerts; trying to finish a good book; or volunteering—especially at events that promote diversity in technology. The accomplishment she is most proud of is that both of her awesome daughters are engineers!

Automate or Not-o-Mate?, page 185

Kasun Indrasiri

Kasun Indrasiri is the coauthor of *gRPC Up & Running* (O'Reilly, 2020) and *Microservices for the Enterprise* (Apress, 2018), and a product manager/senior director at WSO2. He is also a committer and PMC member at the Apache Software Foundation and the founder of the Bay Area Microservices, APIs, and Integration meetup group.

Integrating Microservices in Cloud Native Architecture, page 29

Katie McLaughlin

Katie McLaughlin (*@glasnt*) has worn many hats over the years. She has been a software developer for many languages, systems administrator for multiple operating systems, and speaker on many topics. She is currently a developer advocate at Google Cloud, and was a recipient of the 2017 O'Reilly Open Source Award. When she's not changing the world, she enjoys cooking, making tapestries, and seeing just how well various application stacks handle emoji.

Containers Aren't Magic, page 32

Ken Broeren

Ken Broeren is a coach, instructor, and consultant who teaches technologists how to be more effective by helping them become more human. He's a 25-year veteran of the IT industry and has held a variety of engineering and management roles in the military, finance, publishing, and architecture/engineering/construction industries. He lives in the Colorado mountains about an hour outside of Denver.

The Power of Vulnerability, page 105

Ken Corless

Ken Corless, DXC Technology's executive vice president of Technology, Offerings & Partners, is responsible for DXC's technology strategy and driving innovation and growth in the company's core offerings. He has more than 30 years' experience in client-facing roles in the IT industry. Previously, he served as chief technology officer for Deloitte's cloud practice, and he spent more than 28 years at Accenture, serving as Technology Global managing partner, executive director of Enterprise Applications, and managing director of Technology & Architecture. Based in Chicago, he is a graduate of the Massachusetts Institute of Technology, where he received one BS in management and a second in mathematics.

Managing Network Transit Costs in the Cloud, page 171
The Cloud Is Not About the Cloud, page 220

Kendall Miller

Kendall Miller was the first hire at Fairwinds (back when it was ReactiveOps) and has been helping companies adopt cloud native infrastructure for the last five years. Today his company focuses on Kubernetes enablement, and he spends his time in a rocking chair, with a cane, yelling loudly about semi-important things with his team.

Your CIO Wants to Replatform Only Once, page 34

Kim Schlesinger

Kim Schlesinger is a site reliability engineer. Prior to being an SRE, Kim was an instructional designer at a code school, and before that an elementary school special education teacher. Kim loves working at the intersection of tech and adult education. You can follow her at *kimschlesinger.com* or on Twitter (*@kimschles*).

Practice Visualizing Distributed Systems, page 36

Kit Merker

Kit Merker heads business development for Nobl9, driving early-stage growth in service reliability for modern cloud native developers. In his 20+ years of experience with large-scale software development projects he has worked in a variety of roles, from coding to engineering manager, evangelism to product management. Prior to Nobl9, Kit helped grow JFrog into a billion-dollar company and worked as a product manager for Kubernetes and related container initiatives for Google Cloud. Before that, he spent 10 years at Microsoft, where he worked on several products—Windows, Azure, Office 365, and Bing. Twitter: *@KitMerker*.

The Basics of Service-Level Objectives, page 107

Laura Santamaria

Laura Santamaria loves to learn and explain how things work. In her job as a developer advocate at LogDNA, she bridges the gap between external developers and SREs and LogDNA's internal engineering teams. Also, she curates educational content like "A Minute on the Mic" and the "Logger to Logger" newsletter. Find her on Twitter at *@nimbinatus*, on GitHub as *nimbinatus*, and on her website at *nimbinatus.com*. When not at work, she cohosts Austin DevOps and Cloud Austin, taught Python for Women Who Code Austin for many years, and volunteers with devopsdays Austin. She enjoys mucking around with open hardware and has a particular affinity for projects that have blinky lights. Outside of tech, Laura runs, plays with her dogs, throws discs, and watches clouds—the real kind.

Oh, No: No Logs, page 110

Laziz Turakulov

Laziz Turakulov is a digital business analyst at BAT. He has more than 20 years of IT experience as a developer, system administrator, and solution architect with exposure to the operations, marketing, R&D, and now also digital innovation business areas. Laziz holds various certifications from Google (GCP Associate Cloud Engineer), Microsoft (Azure DevOps Engineer Expert, Azure AI Engineer Associate, Azure IoT Developer Specialty, MCSE, MCSD), SAP, Lotus Development, and Sun Microsystems. In his free time

between full-time work and preparation for the next IT certification exam, Laziz is learning the art of 3D modeling on the Blender open source platform, practicing tonal pronunciation in Mandarin Chinese (to prepare for the next level, HSK2), and exploring principles of Japanese swordsmanship at the local iaido dojo.

Stay Curious, page 232

Lee Atchison

Lee Atchison is a recognized industry thought leader in cloud computing and the author of the bestselling book *Architecting for Scale* (O'Reilly, 2020), currently in its second edition. Lee has 33 years of industry experience. He spent eight years at New Relic, where he led the construction of a solid service-based system architecture and processes that allowed scaling from a start-up to a high-traffic public enterprise. He also spent seven years at Amazon, where he led the creation of the company's first software download store, created AWS Elastic Beanstalk, and managed the migration of Amazon's retail platform to a new service-based architecture. Lee has consulted with leading organizations on how to modernize their application architectures and transform their organizations at scale. He is widely quoted in publications as an industry expert and has been a featured speaker at events across the globe.

Failing a Cloud Migration, page 61

Lisa Huynh

Lisa Huynh is a lead software engineer at Storyblocks, making data better. She holds an MS in computer science from George Mason University in Fairfax, VA. For over eight years she's been working her way across the tech stack, from battling browser quirks to carving out microservices. When not fiddling with code, you'll probably find her hanging out on aerial silks or buried in books. Find her on Twitter at *@nomnomlisa*.

Know Where to Scale, page 39
Use Checklists to Manage Risk, page 112

Lukas Ruebbelke

Lukas Ruebbelke is the vice president of developer growth at BrieBug, where he has the greatest job in the world. Lukas gets to spend all his time mentoring and training developers to be effective and build things that people care about. He's also a Google Developer Expert, published author, conference speaker, and event organizer.

Everything Is Just Ones and Zeros, page 150

Manasés Jesús Galindo Bello

Manasés Jesús Galindo Bello has been developing applications since high school, employing various programming languages, tech stacks, and software development methodologies. He has architected and implemented distributed systems and cloud native applications, as well as led projects and delivered working software to international corporations in the banking, IT, and IoT sectors (e.g., HSBC, HP, IBM, and Software AG). He has published articles, spoken at international conferences, and trained fellow software engineers. When not doing tech-related stuff, he enjoys exploring new cities, contemplating on the beach, and playing the saxophone. More details can be found on his personal website, *manasesjesus.com*.

Serverless Bad Practices, page 41

Manjeet Dadyala

Manjeet Dadyala is a strategist with a keen ability to map an organization's desired business outcomes to technology as his defining differentiator. With a record of consulting expertise focused on cloud platforms, business model disruption, and enabling organizations to transform their capabilities into service market opportunities, Manjeet has made his mark with a range of customers, stakeholders, and organizations. He's a technology leader who has not only led teams but defined and developed offerings to serve the evolving needs of tomorrow's customers. Manjeet enjoys working out, eating healthy, investing, and spending time with his family and friends.

Managing the Cloud Migration Cost Spike, page 173

Marcello Marrocos

Marcello Marrocos is passionate about DevOps and a cloud advocate. He's Microsoft Azure DevOps certified, with more than 20 years of experience developing and architecting solutions. With extensive experience in project management and Agile methodologies, he has been applying DevOps principles for many years. Marcello previously worked for consulting companies including Avanade and Accenture. He is currently working for the InterAmerican Development Bank in Washington, DC, where he believes that his work is part of a bigger effort to improve lives in Latin American countries.

Beyond the Portal: Manage Your Cloud with the CLI, page 187

Marko Sluga

Marko Sluga has 20 years of experience in IT and has had the benefit of witnessing the rise of cloud computing. He has worked on a variety of cloud-related projects, from corporate virtualization to migrations to DevOps, as well as fully automated, intelligent, serverless, and cloud native solutions. He is an AWS certified instructor and has authored three books on AWS.

Getting Started with AWS Lambda, page 43

Mattias Geniar

Mattias Geniar is a software developer, system administrator, and indie hacker. He's currently building the Oh Dear monitoring service. You can follow him on Twitter at *@mattiasgeniar*.

It's OK if You're Not Running Kubernetes, page 46

Michael Friedrich

Michael Friedrich is a developer evangelist with more than 15 years of experience in ops and infrastructure management. He is passionate about open source development (C++, C#, Go) and enjoys talking about CI/CD, monitoring/observability, and security at events and meetups. Currently, Michael is working at GitLab. When he is not engaging on social media, Michael enjoys building LEGO models.

Everything Is a DNS Problem: How to (Im)prove, page 114

Michael Winslow

Michael Winslow picked up his love for programming when he was 10 years old writing GW-Basic code on his Tandy-1000. With his passion for designing simple solutions to complex problems, Michael has played key roles at companies including Aramark, Ortho-McNeil, Oracle, and Xfinity Mobile. He is a seasoned international public speaker who enjoys using his platform to uplift engineers and create powerfully diverse teams in technology. Michael is currently a DevOps advocate, Agile enthusiast, and dedicated people-leader.

Damn It, Jim! I'm a Cloud Engineer, Not an Accountant!, page 175

Michelle Brenner

Michelle Brenner is a senior software engineer with 10 years of experience in tech, from engineering support to manager. She runs an interview-format tech podcast called *From the Source* that examines what tech jobs are really like. A Philadelphia native who now calls Los Angeles home, she is an art school graduate and a self-taught engineer. She enjoys making it easier for others, from artists to entrepreneurs, to create great things. Michelle works to promote diversity and inclusion in tech through conference speaking and organizing, mentoring, board membership, and making sure everyone knows they belong here. You can find her on Twitter: *@michellelynneb*.

Why Every Engineer Should Be a Cloud Engineer, page 15

Mike Kavis

Mike Kavis has served in numerous technical roles such as CTO, chief architect, and VP positions and has over 30 years of experience in software development and architecture. A pioneer in cloud computing, Mike led a team that built the world's first high-speed transaction network in Amazon's public cloud, won the 2010 AWS Global Startup Challenge, and is ranked as one of the Top 100 Cloud Experts and Influencers. Mike is the author of *Architecting the Cloud: Design Decisions for Cloud Computing Service Models (IaaS, PaaS, SaaS)* (Wiley, 2014) and the forthcoming *Accelerating Cloud Adoption: Optimizing the Enterprise for Speed and Agility* (O'Reilly, 2021).

Optimizing Processes for the Cloud: Patterns and Antipatterns, page 63
The Cloud Is Bigger than IT: Enterprise-Wide Training Strategies, page 222

Mike Silverman

Mike Silverman, Head of Strategy at the Financial Services Information Sharing and Analysis Center (FS-ISAC), has a unique blend of business and technology backgrounds, with more than 20 years of experience combined in technology leadership and management consulting across many industries. He enables firms to innovate, scale, and transform through increasing productivity, reducing costs, and streamlining processes and operations. You can reach Mike on Twitter (*@mikebsilverman*) and LinkedIn (*linkedin.com/in/mikebsilverman*).

Why the Lift-and-Shift Model Is Unlikely to Succeed, page 66

Nathen Harvey

Nathen Harvey, cloud developer advocate at Google, helps the community understand and apply DevOps and SRE practices in the cloud. He is part of the global organizing committee for the devopsdays conference series and was a technical reviewer for the Accelerate State of DevOps Report. Nathen formerly led the Chef community, cohosted the Food Fight Show, and managed operations and infrastructure for a diverse range of web applications.

What Is the Cloud?, page 2
Why the Cloud?, page 4

Nikhil Nanivadekar

Nikhil Nanivadekar is the active project lead for the open source Eclipse Collections framework. He has been working in the financial sector as a Java developer since 2012. Nikhil holds a bachelor's degree in mechanical engineering from the University of Pune, and a master's of science in mechanical engineering with a specialization in robotics and controls from the University of Utah. In 2018, Nikhil was selected as a Java Champion. In 2020, he contributed to *97 Things Every Java Programmer Should Know* (O'Reilly). He has always been passionate about open source software, and enjoys creating content and sharing it with others. He has hosted workshops and talks about robotics and data structures, and has given introductory talks to share his enthusiasm about various technologies. He is a regular speaker at technical conferences worldwide. Nikhil is dedicated to providing and enabling learning opportunities for children, and he regularly hosts workshops at conferences such as JCrete4Kids, JavaOne4Kids, OracleCodeOne4Kids, and Devoxx4Kids.

Know Thy Topology, page 48
What's the Time?, page 116

Nirmal Mehta

Nirmal Mehta is a chief technologist in the Strategic Innovations Group at Booz Allen Hamilton, specializing in research, implementation, and integration of emerging technologies for Booz Allen's client base. He leads the firm's efforts in digital research and development, emerging technology strategy, and cloud-based innovation. In addition, he is a containerization subject matter expert (Docker Captain) and advocate for DevOps practices.

Empathy as Code, page 234

Noah Abrahams

Noah Abrahams has been involved in cloud computing, wearing many hats, since around 2007. He has seen all its permutations, from the IaaS versus PaaS versus SaaS debates, to the on-demand bare-metal versus VMs era, and on to microservices, containers, serverless, and FaaS. He's spent the entire time bouncing between low-level hands-on work and higher-level enablement, education, sales, consulting, and architecture. Currently acting as an ambassador for the CNCF, in addition to his day job, he's always happy to talk about the history of the cloud, the associated ecosystem, and emerging patterns. If you find him online or at a conference, he'll definitely make time to talk to you. He can be found on Twitter at *@Noah_Abrahams*.

System Fundamentals Will Still Bite You, page 51

Ori Cohen

Dr. Ori Cohen has a PhD in computer science with a focus on machine learning and the brain–computer interface. He has led a data science team in a smart city start-up, primarily doing natural language processing and natural language understanding research using machine and deep learning. Currently, he is a lead data scientist at New Relic TLV in the field of AIOps. He regularly writes about managing, processes, and all things data science on Medium.com.

Monitor Your Model Dependencies!, page 118

Peter McCool

Peter McCool was raised by quantum mechanics and so came to regard computers as a tool to do one's dirty work from a young age. He started his IT career in 1995, in desktop support. Since then, he's been a Unix system administrator, various kinds of developer, a tester, and a solution architect, among other things. DevOps, as something that involves all of these things and then some, fascinates him.

There's No Such Thing as a Development Environment, page 120

Rachel Sweeney

Rachel Sweeney is an DevOps engineer at the Pew Research Center. She enjoys solving problems using AWS, Python, Kubernetes, and other amazing tools to make data more accessible and reliable to the data scientists she works with. When she's not solving problems for work, she enjoys working on her boat and spending weekends with her wife at a quiet anchorage.

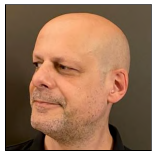
From Zero to Cloud Engineer in Less Than a Year, page 236

Reza Salari

Reza Salari is a technology executive in the financial services/insurance industry and is passionate about the cloud and how it opens up new possibilities for businesses. He is an AWS Academy Accredited Educator for the University of Maryland and has taught undergraduate and graduate cybersecurity and cloud courses for the past five years. He has spent 10 years of his career working abroad for both defense and private sector clients, with a strong focus on training and awareness and people strategy.

Managing Up: Engaging with Executives on the Cloud, page 17

Ricardo Miranda



Ricardo Miranda is a mechanical engineer who started his career doing high-performance computing in oceanography. Since then, his passion for large-scale problems has only grown. Distributed systems in cloud environments are his current obsession. When he's not staring at a screen, you may find him having fun playing soccer or riding a mountain bike.

Be Prepared to Repeat, page 152

Rustem Feyzkhanov

Rustem Feyzkhanov is a machine learning engineer at Instrumental, where he creates analytical models for the manufacturing industry, and an AWS Machine Learning Hero. Rustem is passionate about serverless infrastructure (and AI deployments on it) and is the author of the course and book *Serverless Deep Learning with TensorFlow and AWS Lambda* (Packt Publishing, 2019) and the video course *Practical Deep Learning on the Cloud*.

Cloud Processing Is Not About Speed, page 53

Ryan Bell

Ryan Bell is the founder of a small creative agency in Las Vegas. He has always had a deep passion for exploring the intersections between art, science, technology, and the humanities. This has taken him down the seemingly divergent paths of studying aesthetic design, functional programming, music composition, machine learning, entrepreneurship, biochemistry, philanthropy, and philosophy. Looking back, it's all converged to lead him to this point.

Your Greatest Products Are Not the Applications and Services You Produce, page 154

Ryan Frantz

Ryan Frantz has been practicing software development and web operations for over 20 years, holding roles in engineering and management. His experiences were gained at companies ranging from start-ups to large organizations, covering a breadth of industries including healthcare, ecommerce, and fintech. He is a craftsman and a storyteller.

Incident Analysis and Chaos Engineering: Complementary Practices, page 122

Sarah Cecchetti

Sarah Cecchetti is a principal product manager at AWS Identity. She cofounded a professional organization for identity practitioners called IDPro and coauthored NIST 800-63-C, "Digital Identity Guidelines." She was named one of the top 100 influencers in identity. She has spoken on information security at the RSA Conference, and keynoted Identiverse. Sarah has been quoted as an industry expert in *The LA Times*, *Forbes*, and *Wired*.

You Can't Get Information Security Right Without Getting Identity Right, page 85

Scott Pantall

Scott Pantall is a father, husband, hockey fan, tech blogger, comic/art n00b, and full-stack software engineer at Infinicept, a payments tech startup in Denver, CO. Infinicept hosts everything in the cloud using Azure. Scott has helped grow his team from 5 employees to over 40 employees in just over two years. He enjoys making working with his team as productive and enjoyable as possible for everyone involved. Twitter: *@scottpantall*.

Effectively Monitoring Cloud Services Requires Planning, page 177

Shayon Mukherjee

Shayon Mukherjee is an infrastructure engineer with a former background in product engineering. A cloud native, Shayon enjoys building scalable infrastructure and systems at companies like Intercom and similar high-growth startups. He takes a deep interest in performance, availability, and operations, and believes on-call experience plays a major role in career growth. In his spare time, he loves to read and spend time outdoors. He can be found on Twitter as *@shayonj*.

Handling Network Failures in the Cloud, page 206

Simon Aronsson

Simon Aronsson is a 30-something gopher, developer, public speaker, and meetup organizer from Sweden. He has been working in tech for the last 10 years or so, in roles ranging from full-stack dev and systems architect to scrum master and ops engineer. During the last couple of years, he has put a lot of his time into DevOps practices, cloud development, automation, and creating highly efficient, self-organizing teams. In his spare time, you'll usually find Simon either out and about on his longboard or alpine skis, caring for the chilies in his hydroponic window garden, building software or hardware, or playing with his Commodore 64.

Avoid Big Rewrites, page 156

Stephen Kuenzli

Stephen Kuenzli loves designing, building, and operating software systems that are usable, secure, highly available, and scalable. Stephen founded k9 Security to help cloud teams secure data and manage risk, while delivering change quickly and confidently. He publishes DevOps, cloud, and security content regularly at *https://nodramadevops.com* and coauthored *Docker in Action* (Manning, 2019). You can reach him on Twitter and most tech platforms at *@skuenzli*.

Why Are Good AWS Security Policies So Difficult?, page 87
How Should I Organize My AWS Accounts?, page 125

Theresa Neate

Theresa Neate is a QA practice lead and developer advocate with several years of leadership experience, who loves Lean and Agile and advocates for holistic system quality and systems thinking. She firmly believes that quality is not equal to testing and that quality applies to the entire system life cycle, from inception to her personal favorites, DevOps, operations, and infrastructure. Theresa's background spans more than two decades of technology and leadership experience. In the last decade she has worked at ThoughtWorks, Australia Post's Digital Delivery Centre, and digital icon REA Group, where she spent almost five years; she is now national practice area lead/head of quality engineering for the technology consultancy Slalom Build Australia. She is cofounder of and co-organizer and trainer at DevOpsGirls. She also writes and speaks on the topics of quality, systems thinking, Lean, Agile, and DevOps. Theresa is a lifelong and eternally curious skeptic and learner. You can find her on Twitter: *@theresaneate*.

Lean QA: The QA Evolving in the DevOps World, page 158
Systems Thinking and the Support Pager, page 224

Tidjani Belmansour

Tidjani Belmansour is a cloud solutions architect working for Cofomo Canada (*cofomo.com*), a Microsoft Azure MVP, the co-organizer of the Azure Quebec user group, a book reviewer, a blogger (*dev.to/tidjani* and *espacenuagic.com*), and a speaker at conferences and user groups. Tidjani also holds a PhD in industrial engineering. He likes sharing what he learns, as he truly believes that we learn more by sharing. While some may say "Sharing is caring," he prefers to say "Sharing is learning." You can reach Tidjani on Twitter: *@Tidjani_B*.

Monitor, You Will, page 130

Tiffany Jachja

Tiffany Jachja is a technical evangelist at Harness. She is an advocate for better software delivery, sharing applicable practices, stories, and content around modern technologies. Before joining Harness, Tiffany was a consultant at Red Hat. There she used her experience to help customers build software applications living in the cloud. Twitter: *@tiffanyjachja*.

Source Code Management for Software Delivery, page 161
Curating a DevOps Culture and Experience, page 226

Wietse Venema

Wietse Venema is a software engineer. If he's not training teams to build scalable and reliable software, he's figuring out how things work so he can be a better engineer and teacher. He authored *Building Serverless Applications with Google Cloud Run* (O'Reilly, 2020), and he's proud to be the name twin (not family) of the famous Wietse Venema who wrote Postfix.

How Serverless Simplifies the Developer Experience, page 55

Will Deane

Will Deane has been working in what is currently known as cyber security for over 20 years, in a range of technical roles including security operations, security engineering, security testing (penetration testing), and technical security architecture. Over this time he's worked for Cable & Wireless Worldwide and Regency IT Consulting, a boutique security consultancy where as chief technology officer he was responsible for managed security services in addition to technical security consulting. He is currently an independent consultant through his own company, ASX Consulting Limited, predominantly helping public sector organizations safely move sensitive workloads to the public cloud. He's also an experienced cyber security trainer, regularly delivering British Computer Society (BCS) and National Cyber Security Centre (NCSC) certified courses in security architecture and cloud security as well as developing bespoke courses for clients.

Side Channels and Covert Communications in Cloud Environments, page 90

Zach Thomas

Zach Thomas leads the cloud SRE team for Genesys. He is fascinated by the way complex human systems influence complex technology systems, and vice versa. Zach has 20 years of experience building information systems for the web, in the domains of education, collaboration, and telecommunications. In a past life he was a founding member of Okkervil River, quite a good indie rock band.

Reliable Systems Don't Happen by Accident, page 133

Zachary Nickens

Zachary Nickens, currently a site reliability engineer at Woolpert, is a Google Professional Cloud Architect with expertise in Kubernetes, infrastructure as code, site reliability engineering, CI/CD, and Linux systems engineering. He has deep expertise in spatially enabled data products, data modeling, and Bayesian statistics for analysis and prediction. Zac formerly worked with the Planning and Integration Directorate, Geospatial Integration Office of the Air Force Civil Engineer Center (AFCEC) at Lackland AFB at Joint Base San Antonio and Tyndall AFB in Panama City. He also worked as the staff data engineer for data pipelines, statistical modeling, and automation systems at USAF GeoBase JBSA. Zac is passionate about the human elements of DevOps, geospatial software, autism awareness, and golf, which he routinely tweets about at *@the_nickens*.

What Is Toil, and Why Are SREs Obsessed with It?, page 135
Treat Your Infrastructure like Software, page 190

Index

B

C

D

E

F

G

H

M

N

O

P

Q

R

S

T