O'REILLY®

97 Things Every Cloud Engineer Should Know

Collective Wisdom from the Experts

Edited by Emily Freeman & Nathen Harvey

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

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

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

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.

978-1-492-07673-5

[LSI]

Table of Contents

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

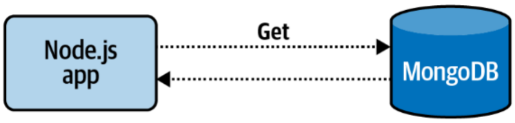
Site Reliability Engineer at Fairwinds

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.

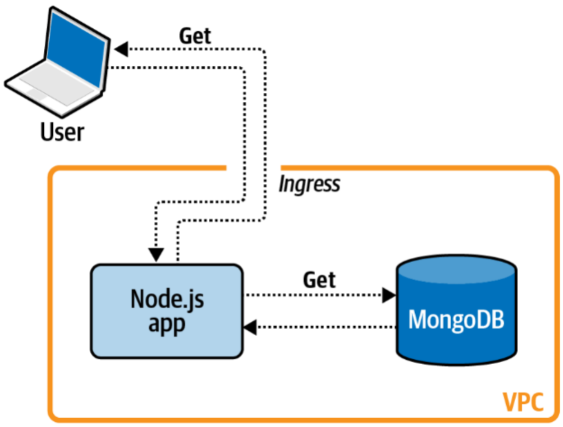
Highly skilled cloud engineers are able to imagine parts of the systems they build and maintain, and they can visualize how a request flows from one component to another. This isn't a skill most of us are born with, but with determination and practice, you can begin imagining and understanding your invisible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