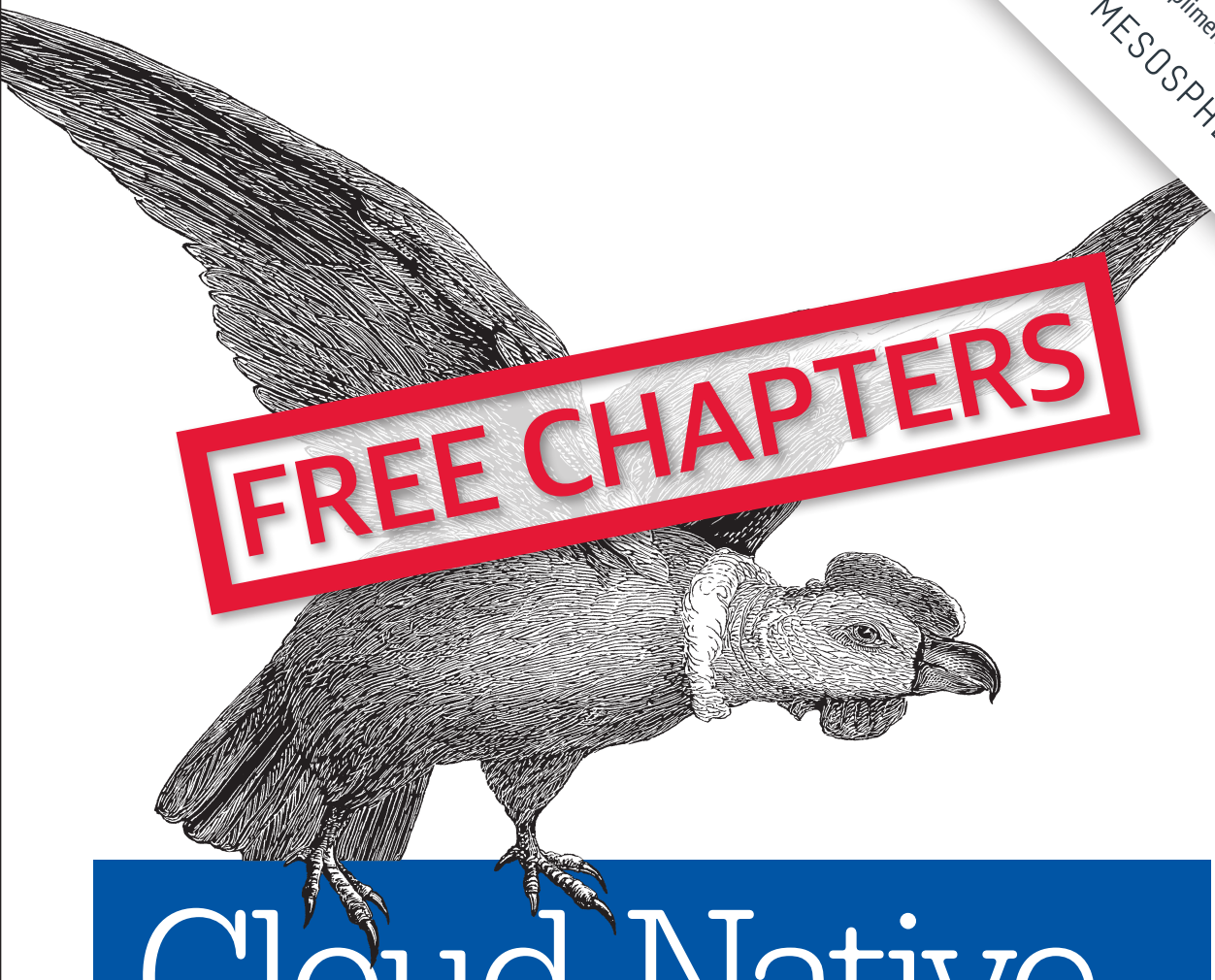
O'REILLY®

Compliments of MESOSPHERE



Cloud Native Infrastructure

PATTERNS FOR SCALABLE INFRASTRUCTURE AND APPLICATIONS IN A DYNAMIC ENVIRONMENT

Justin Garrison & Kris Nova

O'REILLY®

Cloud Native Infrastructure

Cloud native infrastructure is more than servers, network, and storage in the cloud—it is as much about operational hygiene as it is about elasticity and scalability. In this book, you'll learn practices, patterns, and requirements for creating infrastructure that meets your needs—one capable of managing the full life cycle of cloud native applications.

Justin Garrison and Kris Nova reveal hard-earned lessons on architecting infrastructure from companies such as Google, Amazon, and Netflix. They draw inspiration from projects adopted by the Cloud Native Computing Foundation (CNCF) and provide examples of patterns seen in existing tools such as Kubernetes.

With this book, you will:

- Understand why cloud native infrastructure is necessary to effectively run cloud native applications
- Use guidelines to decide when—and if—your business should adopt cloud native practices
- Learn patterns for deploying and managing infrastructure and applications
- Design tests to prove that your infrastructure works as intended, even in a variety of edge cases
- Learn how to secure infrastructure with policy as code

Justin Garrison is an engineer at one of the world's largest media companies. He loves community and open source, and he strives to address people's needs instead of fixing their problems.

Kris Nova is a Senior Developer Advocate for Microsoft with an emphasis in containers and the Linux operating system. She is also a backend infrastructure engineer and a Kubernetes maintainer.

"The on-demand nature of cloud native architecture resets our assumptions about how to rapidly build efficient, scalable, and reliable systems. With this book, Kris and Justin have written an excellent guide to the new principles and practices of cloud native."

—Adrian Cockcroft

"Cloud native computing is the way all software will be deployed over the next decade. Kris and Justin have been leaders in the cloud native community and I'm thrilled to see their work helping others navigate this new and exciting ecosystem."

—Dan Kohn
Executive Director, Cloud Native Computing Foundation

9 781492 036098

MESOSPHERE **DC/OS**

THE PREMIERE PLATFORM FOR CONTAINERS & FAST DATA

Kubernetes available now on Mesosphere DC/OS

Delivering point-and-click simplicity for building, deploying, operating, and scaling data-intensive applications on any infrastructure—public and on-prem with hybrid portability.

- **1-click install & operation of 100+ services**
- **Improved efficiency cuts infrastructure costs**
- **Highly available and secure by default**
- **100% pure up-stream Kubernetes**

LEARN MORE →

> Mesosphere provides customers with a robust platform for building, deploying, and operating data-rich, containerized applications.

- Allan Naim, Kubernetes Product Manager, Google

Cloud Native Infrastructure

Patterns for Scalable Infrastructure and Applications in a Dynamic Environment

This Excerpt contains Chapters 1–4, 7 of *Cloud Native Infrastructure*. The final book is available for sale on oreilly.com and through other retailers.

Justin Garrison and Kris Nova

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Cloud Native Infrastructure
by Justin Garrison and Kris Nova

Printed in the United States of America.

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

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

Editors: Virginia Wilson and Nikki McDonald
Production Editor: Kristen Brown
Copyeditor: Amanda Kersey
Proofreader: Rachel Monaghan
Indexer: Angela Howard
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest
Tech Reviewers: Peter Miron, Andrew Schafer, and Justice London

November 2017: First Edition

Revision History for the First Edition
2017-10-25: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491984307* for release details.

This work is part of a collaboration between O'Reilly and Mesosphere. See our *statement of editorial independence*.

978-1-492-03609-8

[LSI]

Table of Contents

Foreword

Today's economy is being shaped by apps using millions of gigabytes of data to deliver unique experiences an individual at a time.

Economic growth is currently dominated by companies transforming how we live our lives with highly personalized, data-informed, real-time interactions. For the first time in recent history, seven of the ten most valuable companies in the world are Internet giants. Their success comes partly from being able to develop globally scalable application infrastructure to provide individualized experiences that support and fulfill, surprise and delight, or inform and entertain. To regain a competitive edge, most traditional enterprises I talk to want to emulate these capabilities using the same tools that power the Internet giants, but balk because they find defining the requirements needed to support those solutions hard, using the needed cloud native infrastructure.

Many of the patterns and tools developed and used by these technology companies have been democratized through open-source and cloud tools. As examples, Google designed Kubernetes using their decade-long experience with container orchestration, and Facebook built the Cassandra distributed database to provide massive backend scale. On the management side, Twitter showed the world how to improve user experience using Apache Mesos to pool and automate running containerized data-intensive applications. All of the once-closed capabilities of the Silicon Valley elite are now open source and widely available and supported. Given the right "know how," anyone can now leverage these tools to tap into the near limitless capacity of cloud native applications.

The excerpts from this book will help you design, build, and manage the architecture needed to successfully support cloud native applications. You'll also find guidance to help you understand when to build your own services and when to choose public cloud managed options. One of the most difficult challenges for operators is managing the complexity of the many different open-source services, configurations, and versions needed to build cloud native applications. This challenge is compounded

when services are required to run across any cloud, any infrastructure. For this reason, we've included a chapter that helps IT professionals learn the best practices needed to manage cloud native infrastructure.

Mesosphere is excited to partner with O'Reilly to offer this excerpt, as it aligns with our mission to make it insanely easy to build and elastically scale data-intensive, modern applications on any infrastructure. Our platform, DC/OS, provides an easy-to-use platform to deploy and scale popular cloud native application infrastructure components, including container orchestration with Kubernetes, data services such as Apache Kafka and Apache Spark, machine learning frameworks like Tensorflow, traditional management software like Jenkins, and 100+ popular tools.

We hope you enjoy this excerpt, and consider Mesosphere DC/OS to jump start your journey towards building, deploying, and scaling the data-intensive applications that will drive your business towards success.

You can learn more about DC/OS on mesosphere.com.

— Tobi Knaup
Chief Technology Officer, Mesosphere

CHAPTER 1
What Is Cloud Native Infrastructure?

Infrastructure is all the software and hardware that support applications.[1] This includes data centers, operating systems, deployment pipelines, configuration management, and any system or software needed to support the life cycle of applications.

Countless time and money has been spent on infrastructure. Through years of evolving the technology and refining practices, some companies have been able to run infrastructure and applications at massive scale and with renowned agility. Efficiently running infrastructure accelerates business by enabling faster iteration and shorter times to market.

Cloud native infrastructure is a requirement to effectively run cloud native applications. Without the right design and practices to manage infrastructure, even the best cloud native application can go to waste. Immense scale is not a prerequisite to follow the practices laid out in this book, but if you want to reap the rewards of the cloud, you should heed the experience of those who have pioneered these patterns.

Before we explore how to build infrastructure designed to run applications in the cloud, we need to understand how we got where we are. First, we'll discuss the benefits of adopting cloud native practices. Next, we'll look at a brief history of infrastructure and then discuss features of the next stage, called "cloud native," and how it relates to your applications, the platform where it runs, and your business.

Once you understand the problem, we'll show you the solution and how to implement it.

1 Some of these things may be considered applications on their own; in this book we will consider applications to be the software that generates revenue to sustain a business.

Cloud Native Benefits

The benefits of adopting the patterns in this book are numerous. They are modeled after successful companies such as Google, Netflix, and Amazon—not that the patterns alone guaranteed their success, but they provided the scalability and agility these companies needed to succeed.

By choosing to run your infrastructure in a public cloud, you are able to produce value faster and focus on your business objectives. Building only what you need to create your product, and consuming services from other providers, keeps your lead time small and agility high. Some people may be hesitant because of "vendor lock-in," but the worst kind of lock-in is the one you build yourself.

Consuming services also lets you build a customized platform with the services you need (sometimes called Services as a Platform [SaaP]). When you use cloud-hosted services, you do not need expertise in operating every service your applications require. This dramatically impacts your ability to change and adds value to your business.

When you are unable to consume services, you should build applications to manage infrastructure. When you do so, the bottleneck for scale no longer depends on how many servers can be managed per operations engineer. Instead, you can approach scaling your infrastructure the same way as scaling your applications. In other words, if you are able to run applications that can scale, you can scale your infrastructure with applications.

The same benefits apply for making infrastructure that is resilient and easy to debug. You can gain insight into your infrastructure by using the same tools you use to manage your business applications.

Cloud native practices can also bridge the gap between traditional engineering roles (a common goal of DevOps). Systems engineers will be able to learn best practices from applications, and application engineers can take ownership of the infrastructure where their applications run.

Cloud native infrastructure is not a solution for every problem, and it is your responsibility to know if it is the right solution for your environment (see Chapter 2). However, its success is evident in the companies that created the practices and the many other companies that have adopted the tools that promote these patterns.

Before we dive into the solution, we need to understand how these patterns evolved from the problems that created them.

Servers

At the beginning of the internet, web infrastructure got its start with physical servers. Servers are big, noisy, and expensive, and they require a lot of power and people to keep them running. They are cared for extensively and kept running as long as possible. Compared to cloud infrastructure, they are also more difficult to purchase and prepare for an application to run on them.

Once you buy one, it's yours to keep, for better or worse. Servers fit into the well-established capital expenditure cost of business. The longer you can keep a physical server running, the more value you will get from your money spent. It is always important to do proper capacity planning and make sure you get the best return on investment.

Physical servers are great because they're powerful and can be configured however you want. They have a relatively low failure rate and are engineered to avoid failures with redundant power supplies, fans, and RAID controllers. They also last a long time. Businesses can squeeze extra value out of hardware they purchase through extended warranties and replacement parts.

However, physical servers lead to waste. Not only are the servers never fully utilized, but they also come with a lot of overhead. It's difficult to run multiple applications on the same server. Software conflicts, network routing, and user access all become more complicated when a server is maximally utilized with multiple applications.

Hardware virtualization promised to solve some of these problems.

Virtualization

Virtualization emulates a physical server's hardware in software. A virtual server can be created on demand, is entirely programmable in software, and never wears out so long as you can emulate the hardware.

Using a hypervisor[2] increases these benefits because you can run multiple virtual machines (VMs) on a physical server. It also allows applications to be portable because you can move a VM from one physical server to another.

One problem with running your own virtualization platform, however, is that VMs still require hardware to run. Companies still need to have all the people and processes required to run physical servers, but now capacity planning becomes harder because they have to account for VM overhead too. At least, that was the case until the public cloud.

2 Type 1 hypervisors run on hardware servers with the sole purpose of running virtual machines.

Infrastructure as a Service

Infrastructure as a Service (IaaS) is one of the many offerings of a cloud provider. It provides raw networking, storage, and compute that customers can consume as needed. It also includes support services such as identity and access management (IAM), provisioning, and inventory systems.

IaaS allows companies to get rid of all of their hardware and to rent VMs or physical servers from someone else. This frees up a lot of people resources and gets rid of processes that were needed for purchasing, maintenance, and, in some cases, capacity planning.

IaaS fundamentally changed infrastructure's relationship with businesses. Instead of being a capital expenditure benefited from over time, it is an operational expense for running your business. Businesses can pay for their infrastructure the same way they pay for electricity and people's time. With billing based on consumption, the sooner you get rid of infrastructure, the smaller your operational costs will be.

Hosted infrastructure also made consumable HTTP Application Programming Interfaces (APIs) for customers to create and manage infrastructure on demand. Instead of needing a purchase order and waiting for physical items to ship, engineers can make an API call, and a server will be created. The server can be deleted and discarded just as easily.

Running your infrastructure in a cloud does not make your infrastructure cloud native. IaaS still requires infrastructure management. Outside of purchasing and managing physical resources, you can—and many companies do—treat IaaS identically to the traditional infrastructure they used to buy and rack in their own data centers.

Even without "racking and stacking," there are still plenty of operating systems, monitoring software, and support tools. Automation tools[3] have helped reduce the time it takes to have a running application, but oftentimes ingrained processes can get in the way of reaping the full benefit of IaaS.

Platform as a Service

Just as IaaS hides physical servers from VM consumers, *platform as a service* (PaaS) hides operating systems from applications. Developers write application code and define the application dependencies, and it is the platform's responsibility to create the necessary infrastructure to run, manage, and expose it. Unlike IaaS, which still

3 Also called "infrastructure as code."

requires infrastructure management, in a PaaS the infrastructure is managed by the platform provider.

It turns out, PaaS limitations required developers to write their applications differently to be effectively managed by the platform. Applications had to include features that allowed them to be managed by the platform without access to the underlying operating system. Engineers could no longer rely on SSHing to a server and reading log files on disk. The application's life cycle and management were now controlled by the PaaS, and engineers and applications needed to adapt.

With these limitations came great benefits. Application development cycles were reduced because engineers did not need to spend time managing infrastructure. Applications that embraced running on a platform were the beginning of what we now call "cloud native applications." They exploited the platform limitations in their code and in many cases changed how applications are written today.

12-Factor Applications

Heroku was one of the early pioneers who offered a publicly consumable PaaS. Through many years of expanding its own platform, the company was able to identify patterns that helped applications run better in their environment. There are 12 main factors that Heroku defines that a developer should try to implement.

The 12 factors are about making developers efficient by separating code logic from data; automating as much as possible; having distinct build, ship, and run stages; and declaring all the application's dependencies.

If you consume all your infrastructure through a PaaS provider, congratulations, you already have many of the benefits of cloud native infrastructure. This includes platforms such as Google App Engine, AWS Lambda, and Azure Cloud Services. Any successful cloud native infrastructure will expose a self-service platform to application engineers to deploy and manage their code.

However, many PaaS platforms are not enough for everything a business needs. They often limit language runtimes, libraries, and features to meet their promise of abstracting away the infrastructure from the application. Public PaaS providers will also limit which services can integrate with the applications and where those applications can run.

Public platforms trade application flexibility to make infrastructure somebody else's problem. Figure 1-1 is a visual representation of the components you will need to manage if you run your own data center, create infrastructure in an IaaS, run your applications on a PaaS, or consume applications through software as a service (SaaS).

The fewer infrastructure components you are required to run, the better; but running all your applications in a public PaaS provider may not be an option.

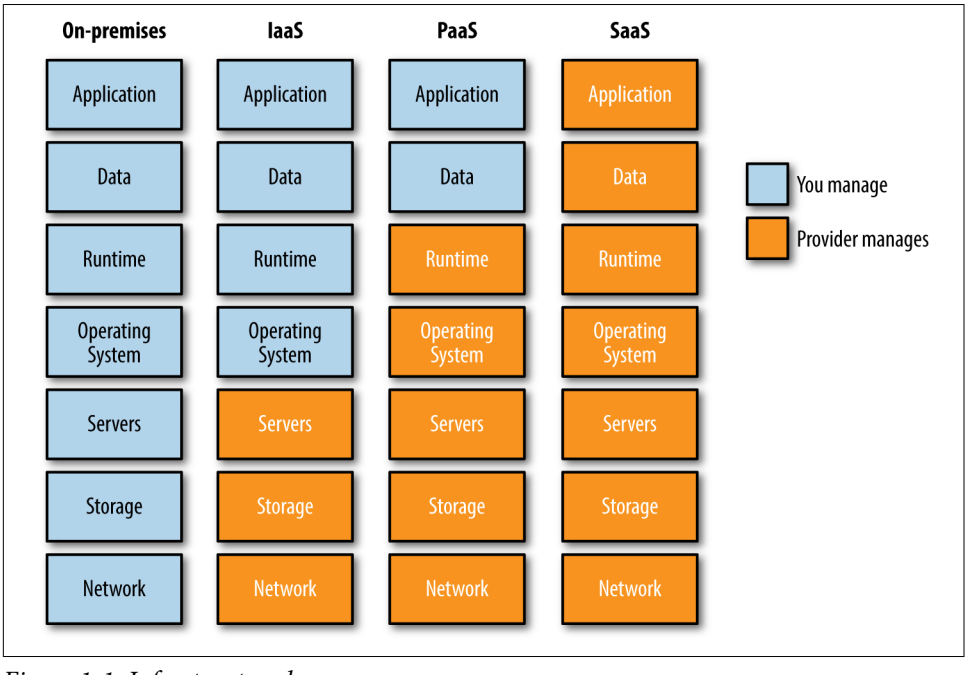


Figure 1-1. Infrastructure layers

Cloud Native Infrastructure

"Cloud native" is a loaded term. As much as it has been hijacked by marketing departments, it still can be meaningful for engineering and management. To us, it is the evolution of technology in the world where public cloud providers exist.

Cloud native infrastructure is infrastructure that is hidden behind useful abstractions, controlled by APIs, managed by software, and has the purpose of running applications. Running infrastructure with these traits gives rise to a new pattern for managing that infrastructure in a scalable, efficient way.

Abstractions are useful when they successfully hide complexity for their consumer. They can enable more complex uses of the technology, but they also limit how the technology is used. They apply to low-level technology, such as how TCP abstracts IP, or higher levels, such as how VMs abstract physical servers. Abstractions should always allow the consumer to "move up the stack" and not reimplement the lower layers.

Cloud native infrastructure needs to abstract the underlying IaaS offerings to provide its own abstractions. The new layer is responsible for controlling the IaaS below it as well as exposing its own APIs to be controlled by a consumer.

Infrastructure that is managed by software is a key differentiator in the cloud. Software-controlled infrastructure enables infrastructure to scale, and it also plays a role in resiliency, provisioning, and maintainability. The software needs to be aware of the infrastructure's abstractions and know how to take an abstract resource and implement it in consumable IaaS components accordingly.

These patterns influence more than just how the infrastructure runs. The types of applications that run on cloud native infrastructure and the kinds of people who work on them are different from those in traditional infrastructure.

If cloud native infrastructure looks a lot like a PaaS offering, how can we know what to watch out for when building our own? Let's quickly describe some areas that may appear like the solution, but don't provide all aspects of cloud native infrastructure.

What Is Not Cloud Native Infrastructure?

Cloud native infrastructure is not only running infrastructure on a public cloud. Just because you rent server time from someone else does not make your infrastructure cloud native. The processes to manage IaaS are often no different than running a physical data center, and many companies that have migrated existing infrastructure to the cloud[4] have failed to reap the rewards.

Cloud native is not about running applications in containers. When Netflix pioneered cloud native infrastructure, almost all its applications were deployed with virtual-machine images, not containers. The way you package your applications does not mean you will have the scalability and benefits of autonomous systems. Even if your applications are automatically built and deployed with a continuous integration and continuous delivery pipeline, it does not mean you are benefiting from infrastructure that can complement API-driven deployments.

It also doesn't mean you only run a container orchestrator (e.g., Kubernetes and Mesos). Container orchestrators provide many platform features needed in cloud native infrastructure, but not using the features as intended means your applications are dynamically scheduled to run on a set of servers. This is a very good first step, but there is still work to be done.

4 Called "lift and shift."

Scheduler Versus Orchestrator

The terms "scheduler" and "orchestrator" are often used interchangeably.

In most cases, the *orchestrator* is responsible for all resource utilization in a cluster (e.g., storage, network, and CPU). The term is typically used to describe products that do many tasks, such as health checks and cloud automation.

Schedulers are a subset of orchestration platforms and are responsible only for picking which processes and services run on each server.

Cloud native is not about microservices or infrastructure as code. Microservices enable faster development cycles on smaller distinct functions, but monolithic applications can have the same features that enable them to be managed effectively by software and can also benefit from cloud native infrastructure.

Infrastructure as code defines and automates your infrastructure in machine-parsible language or domain-specific language (DSL). Traditional tools to apply code to infrastructure include configuration management tools (e.g., Chef and Puppet). These tools help greatly in automating tasks and providing consistency, but they fall short in providing the necessary abstractions to describe infrastructure beyond a single server.

Configuration management tools automate one server at a time and depend on humans to tie together the functionality provided by the servers. This positions humans as a potential bottleneck for infrastructure scale. These tools also don't automate the extra parts of cloud infrastructure (e.g., storage and network) that are needed to make a complete system.

While configuration management tools provide some abstractions for an operating system's resources (e.g., package managers), they do not abstract away enough of the underlying OS to easily manage it. If an engineer wanted to manage every package and file on a system, it would be a very painstaking process and unique to every configuration variant. Likewise, configuration management that defines no, or incorrect, resources is only consuming system resources and providing no value.

While configuration management tools can help automate parts of infrastructure, they don't help manage applications better. We will explore how cloud native infrastructure is different by looking at the processes to deploy, manage, test, and operate infrastructure in later chapters, but first we will look at which applications are successful and when you should use cloud native infrastructure.

Cloud Native Applications

Just as the cloud changed the relationship between business and infrastructure, cloud native applications changed the relationship between applications and infrastructure. We need to see what is different about cloud native compared to traditional applications so we can understand their new relationship with infrastructure.

For the purposes of this book, and to have a shared vocabulary, we need to define what we mean when we say "cloud native application." Cloud native is not the same thing as a 12-factor application, even though they may share some similar traits. If you'd like more details about how they are different, we recommend reading *Beyond the Twelve-Factor App*, by Kevin Hoffman (O'Reilly, 2012).

A cloud native application is engineered to run on a platform and is designed for resiliency, agility, operability, and observability. *Resiliency* embraces failures instead of trying to prevent them; it takes advantage of the dynamic nature of running on a platform. *Agility* allows for fast deployments and quick iterations. *Operability* adds control of application life cycles from inside the application instead of relying on external processes and monitors. *Observability* provides information to answer questions about application state.

Cloud Native Definition

The definition of a cloud native application is still evolving. There are other definitions available from organizations like the CNCF.

Cloud native applications acquire these traits through various methods. It can often depend on where your applications run[5] and the processes and culture of the business. The following are common ways to implement the desired characteristics of a cloud native application:

- Microservices
- Health reporting
- Telemetry data
- Resiliency
- Declarative, not reactive

5 Running an application in a cloud is valid whether that cloud is hosted by a public vendor or run on-premises.

Microservices

Applications that are managed and deployed as single entities are often called *monoliths*. Monoliths have a lot of benefits when applications are initially developed. They are easier to understand and allow you to change major functionality without affecting other services.

As complexity of the application grows, the benefits of monoliths diminish. They become harder to understand, and they lose agility because it is harder for engineers to reason about and make changes to the code.

One of the best ways to fight complexity is to separate clearly defined functionality into smaller services and let each service independently iterate. This increases the application's agility by allowing portions of it to be changed more easily as needed. Each microservice can be managed by separate teams, written in appropriate languages, and be independently scaled as needed.

So long as each service adheres to strong contracts,[6] the application can improve and change quickly. There are of course many other considerations for moving to microservice architecture. Not the least of these is resilient communication.

We cannot go into all considerations for moving to microservices. Having microservices does not mean you have cloud native infrastructure. If you would like to read more, we suggest Sam Newman's *Building Microservices* (O'Reilly, 2015). While microservices are one way to achieve agility with your applications, as we said before, they are not a requirement for cloud native applications.

Health Reporting

> Stop reverse engineering applications and start monitoring from the inside.
>
> —Kelsey Hightower, *Monitorama PDX 2016: healthz*

No one knows more about what an application needs to run in a healthy state than the developer. For a long time, infrastructure administrators have tried to figure out what "healthy" means for applications they are responsible for running. Without knowledge of what actually makes an application healthy, their attempts to monitor and alert when applications are unhealthy are often fragile and incomplete.

To increase the operability of cloud native applications, applications should expose a health check. Developers can implement this as a command or process signal that the

6 Versioning APIs is a key aspect to maintaining service contracts. See *https://cloud.google.com/appengine/docs/standard/go/designing-microservice-api* for more information.

application can respond to after performing self-checks, or, more commonly, as a web endpoint provided by the application that returns health status via an HTTP code.

Google Borg Example

One example of health reporting is laid out in Google's Borg paper:

> Almost every task run under Borg contains a built-in HTTP server that publishes information about the health of the task and thousands of performance metrics (e.g., RPC latencies). Borg monitors the health-check URL and restarts tasks that do not respond promptly or return an HTTP error code. Other data is tracked by monitoring tools for dashboards and alerts on service-level objective (SLO) violations.

Moving health responsibilities into the application makes the application much easier to manage and automate. The application should know if it's running properly and what it relies on (e.g., access to a database) to provide business value. This means developers will need to work with product managers to define what business function the application serves and to write the tests accordingly.

Examples of applications that provide heath checks include Zookeeper's `ruok` command[7] and etcd's HTTP /health endpoint.

Applications have more than just healthy or unhealthy states. They will go through a startup and shutdown process during which they should report their state through their health check. If the application can let the platform know exactly what state it is in, it will be easier for the platform to know how to operate it.

A good example is when the platform needs to know when the application is available to receive traffic. While the application is starting, it cannot properly handle traffic, and it should present itself as not ready. This additional state will prevent the application from being terminated prematurely, because if health checks fail, the platform may assume the application is not healthy and stop or restart it repeatedly.

Application health is just one part of being able to automate application life cycles. In addition to knowing if the application is healthy, you need to know if the application is doing any work. That information comes from telemetry data.

7 Just because an application provides a health check doesn't mean the application is healthy. One example of this is Zookeeper's `ruok` command to check the health of the cluster. While the application will return `imok` if the health command can be run, it doesn't perform any tests for application health beyond responding to the request.

Telemetry Data

Telemetry data is the information necessary for making decisions. It's true that telemetry data can overlap somewhat with health reporting, but they serve different purposes. Health reporting informs us of application life cycle state, while telemetry data informs us of application business objectives.

The metrics you measure are sometimes called *service-level indicators* (SLIs) or *key performance indicators* (KPIs). These are application-specific data that allow you to make sure the performance of applications is within a *service-level objective* (SLO). If you want more information on these terms and how they relate to your application and business needs, we recommend reading Chapter 4 from *Site Reliability Engineering* (O'Reilly).

Telemetry and metrics are used to solve questions such as:

- How many requests per minute does the application receive?
- Are there any errors?
- What is the application latency?
- How long does it take to place an order?

The data is often scraped or pushed to a time series database (e.g., Prometheus or InfluxDB) for aggregation. The only requirement for the telemetry data is that it is formatted for the system that will be gathering the data.

It is probably best to, at minimum, implement the RED method for metrics, which collects rate, errors, and duration from the application.

Rate
: How many requests received

Errors
: How many errors from the application

Duration
: How long to receive a response

Telemetry data should be used for alerting rather than health monitoring. In a dynamic, self-healing environment, we care less about individual application instance life cycles and more about overall application SLOs. Health reporting is still important for automated application management, but should not be used to page engineers.

If 1 instance or 50 instances of an application are unhealthy, we may not care to receive an alert, so long as the business need for the application is being met. Metrics let you know if you are meeting your SLOs, how the application is being used, and

what "normal" is for your application. Alerting helps you to restore your systems to a known good state.

> If it moves, we track it. Sometimes we'll draw a graph of something that isn't moving yet, just in case it decides to make a run for it.
>
> —Ian Malpass, *Measure Anything, Measure Everything*

Alerting also shouldn't be confused with logging. *Logging* is used for debugging, development, and observing patterns. It exposes the internal functionality of your application. Metrics can sometimes be calculated from logs (e.g., error rate) but requires additional aggregation services (e.g., ElasticSearch) and processing.

Resiliency

Once you have telemetry and monitoring data, you need to make sure your applications are resilient to failure. Resiliency used to be the responsibility of the infrastructure. but cloud native applications need to take on some of that work.

Infrastructure was engineered to resist failure. Hardware used to require multiple hard drives, power supplies, and round-the-clock monitoring and part replacements to keep an application available. With cloud native applications, it is the application's responsibility to embrace failure instead of avoid it.

> In any platform, especially in a cloud, the most important feature above all else is its reliability.
>
> —David Rensin, *The ARCHITECHT Show: A crash course from Google on engineering for the cloud*

Designing resilient applications could be an entire book itself. There are two main aspects to resiliency we will consider with cloud native application: design for failure, and graceful degradation.

Design for failure

The only systems that should never fail are those that keep you alive (e.g., heart implants, and brakes). If your services never go down,[8] you are spending too much time engineering them to resist failure and not enough time adding business value. Your SLO determines how much uptime is needed for a service. Any resources you spend to engineer uptime that exceeds the SLO are wasted.

8 As in a service outage, not a component or instance failure.

Two values you should measure for every service should be your your mean time between failures (MTBF) and mean time to recovery (MTTR). Monitoring and metrics allow you to detect if you are meeting your SLOs, but the platform where the applications run is key to keeping your MTBF high and your MTTR low.

In any complex system, there will be failures. You can manage some failures in hardware (e.g., RAID and redundant power supplies) and some in infrastructure (e.g., load balancers); but because applications know when they are healthy, they should also try to manage their own failure as best they can.

An application that is designed with expectations of failure will be developed in a more defensive way than one that assumes availability. When failure is inevitable, there will be additional checks, failure modes, and logging built into the application.

It is impossible to know every way an application can fail. Developing with the assumption that anything can, and likely will, fail is a pattern of cloud native applications.

The best state for your application to be in is healthy. The second best state is failed. Everything else is nonbinary and difficult to monitor and troubleshoot. Charity Majors, CEO of Honeycomb, points out in her article "Ops: It's Everyone's Job Now" that "distributed systems are never *up*; they exist in a constant state of partially degraded service. Accept failure, design for resiliency, protect and shrink the critical path."

Cloud native applications should be adaptable no matter what the failure is. They expect failure, so they adjust when it's detected.

Some failures cannot and should not be designed into applications (e.g., network partitions and availability zone failures). The platform should autonomously handle failure domains that are not integrated into the applications.

Graceful degradation

Cloud native applications need to have a way to handle excessive load, no matter if it's the application or a dependent service under load. One way to handle load is to degrade gracefully. The *Site Reliability Engineering* book describes graceful degradation in applications as offering "responses that are not as accurate as or that contain less data than normal responses, but that are easier to compute" when under excessive load.

Some aspects of shedding application load are handled by infrastructure. Intelligent load balancing and dynamic scaling can help, but at some point your application may be under more load than it can handle. Cloud native applications need to be aware of this inevitability and react accordingly.

The point of graceful degradation is to allow applications to always return an answer to a request. This is true if the application doesn't have enough local compute resources, as well as if dependent services don't return information in a timely manner. Services that are dependent on one or many other services should be available to answer requests even if dependent services are not. Returning partial answers, or answers with old information from a local cache, are possible solutions when services are degraded.

While graceful degradation and failure handling should both be implemented in the application, there are multiple layers of the platform that should help. If microservices are adopted, then the network infrastructure becomes a critical component that needs to take an active role in providing application resiliency.

Availability Math

Cloud native applications need to have a platform built on top of the infrastructure to make the infrastructure more resilient. If you expect to "lift and shift" your existing applications into the cloud, you should check the service-level agreements (SLAs) for the cloud provider and consider what happens when you use multiple services.

Let's take a hypothetical cloud where we run our applications.

The typical availability for compute infrastructure is 99.95% uptime per month. That means every day your instance could be down for 43.2 seconds and still be within the cloud provider's SLA with you.

Additionally, local storage for the instance (e.g., EBS volume) also has a 99.95% uptime for its availability. If you're lucky, they will both go down at the same time, but worst-case scenario they could go down at different times, leaving your instance with only 99.9% availability.

Your application probably also needs a database, and instead of installing one yourself with a calculated possible downtime of 1 minute and 26 seconds (99.9% availability), you choose the more reliable hosted database with 99.95% availability. This brings your application's reliability to 99.85% or a possible daily downtime of 2 minutes and 9 seconds.

Multiplying availabilities together is a quick way to understand why the cloud should be treated differently. The really bad part is if the cloud provider doesn't meet its SLA, it refunds a percentage of your bill in credits for its platform.

While it's nice that you don't have to pay for their outages, we don't know a single business in the world that survives on cloud credits. If your application's availability is not worth more than what credits you would receive if it were down, you should really consider whether you should run the application at all.

Declarative, Not Reactive

Because cloud native applications are designed to run in a cloud environment, they interact with infrastructure and supporting applications differently than traditional applications do. In a cloud native application, the way to communicate with anything is through the network. Many times network communication is done through RESTful HTTP calls, but it can also be implemented via other interfaces such as remote procedure calls (RPC).

Traditional applications would automate tasks through message queues, files written on shared storage, or local scripts that triggered shell commands. The communication method reacted to an event that happened (e.g., if the user clicks submit, run the submit script) and often required information that existed on the same physical or virtual server.

Serverless

Serverless platforms are cloud native and reactive to events by design. A reason they work so well in a cloud is because they communicate over HTTP APIs, are single-purpose functions, and are declarative in what they call. The platform also helps by making them scalable and accessible from within the cloud.

Reactive communication in traditional applications is often an attempt to build resiliency. If the application wrote a file on disk or into a message queue and then the application died, the result of the message or file could still be completed.

This is not to say technologies like message queues should not be used, but rather that they cannot be relied on for the only layer of resiliency in a dynamic and constantly failing system. Fundamentally, the communication between applications should change in a cloud native environment—not only because there are other methods to build communication resiliency, but also because it is often more work to replicate traditional communication methods in the cloud.

When applications can trust the resiliency of the communication, they should stop reacting and start declaring. Declarative communication trusts that the network will deliver the message. It also trusts that the application will return a success or an error. This isn't to say having applications watch for change is not important. Kubernetes' controllers do exactly that to the API server. However, once change is found, they declare a new state and trust the API server and kubelets to do the necessary thing.

The declarative communication model turns out to be more robust for many reasons. Most importantly, it standardizes a communication model and it moves the functional implementation of how something gets to a desired state away from the

application to a remote API or service endpoint. This helps simplify applications and allows them to behave more predictably with each other.

How Do Cloud Native Applications Impact Infrastructure?

Hopefully, you can tell that cloud native applications are different than traditional applications. Cloud native applications do not benefit from running directly on IaaS or being tightly coupled to a server's operating system. They expect to be run in a dynamic environment with mostly autonomous systems.

Cloud native infrastructure creates a platform on top of IaaS that provides autonomous application management. The platform is built on top of dynamically created infrastructure to abstract away individual servers and promote dynamic resource allocation scheduling.

Automation is not the same thing as autonomous. Automation allows humans to have a bigger impact on the actions they take.

Cloud native is about autonomous systems that do not require humans to make decisions. It still uses automation, but only after deciding the action needed. Only when the system cannot automatically determine the right thing to do should it notify a human.

Applications with these characteristics need a platform that can pragmatically monitor, gather metrics, and then react when failures occur. Cloud native applications do not rely on humans to set up ping checks or create syslog rules. They require self-service resources abstracted away from selecting a base operating system or package manager, and they rely on service discovery and robust network communication to provide a feature-rich experience.

Conclusion

The infrastructure required to run cloud native applications is different than traditional applications. Many responsibilities that infrastructure used to handle have moved into the applications.

Cloud native applications simplify their code complexity by decomposing into smaller services. These services provide monitoring, metrics, and resiliency built directly into the application. New tooling is required to automate the management of service proliferation and life cycle management.

The infrastructure is now responsible for holistic resource management, dynamic orchestration, service discovery, and much more. It needs to provide a platform where services don't rely on individual components, but rather on APIs and autonomous systems. Chapter 2 discusses cloud native infrastructure features in more detail.

CHAPTER 2
When to Adopt Cloud Native

Cloud native infrastructure is not for everybody. Any architecture design is a series of trade-offs. Only people familiar with their own requirements can decide what trade-offs are beneficial or detrimental to their needs.

Do not adopt a tool or design without understanding its impact and limitations. We believe cloud native infrastructure has many benefits, but realize it should not be blindly adopted. We would hate to lead someone down the wrong path for their needs.

How can you know if you should pursue architecting with cloud native infrastructure? Here are some questions you can ask to figure out if cloud native infrastructure is best for you:

- Do you have cloud native applications? (See Chapter 1 for application features that can benefit from cloud native infrastructure.)
- Is your engineering team willing and able to write production-quality code that embodies their job functions as software?
- Do you have API-driven infrastructure (IaaS) on-premises or in a public cloud?
- Does your business need faster development cycles or nonlinear people/systems scaling ratios?

If you answered yes to all of these questions, then you will likely benefit from the infrastructure described in the rest of this book. If you answered no to any of these questions, it doesn't mean you cannot benefit from some cloud native practices, but you may need to do more work before you are ready to fully benefit from this type of infrastructure.

It is just as bad to try to adopt cloud native infrastructure before your business is ready as it is to force a solution that is not right. By trying to gain the benefits without doing due diligence, you will likely fail and write off the architecture as flawed or of no benefit. A tried and failed solution will likely be harder to adopt later, no matter if it is the right solution or not, due to past prejudices.

We will discuss some of the areas to focus on when preparing your organization and technology to be cloud native. There are many things to consider but some of the key areas are your applications, the people at your organization, infrastructure systems, and your business.

Applications

Applications are the easiest part to get ready. The design patterns are well established, and tooling has improved dramatically since the advent of the public cloud. If you are not able to build cloud native applications and automatically deploy them through a verified and tested pipeline, you should not move forward with adopting the infrastructure to support them.

Building cloud native applications does not mean you must first have microservices. It does not mean you have to be developing all your software in the newest trending languages. It means you have to write software that can be managed by software.

The only interaction humans should have with cloud native applications is during their development.[1] Everything else should be managed by the infrastructure or other applications.

Another way to know applications are ready is when they need to dynamically scale with multiple instances. Scaling typically implies multiple copies of the same application behind a load balancer. It assumes that applications store state in a storage service (i.e., database) and do not require complex coordination between running instances.

Dynamic application management implies that a human is not doing the work. Application metrics trigger the scaling, and the infrastructure does the right thing to scale the application. This is a basic feature of most cloud environments. Running autoscaling groups doesn't mean you have cloud native infrastructure; but if autoscaling is a requirement, it may indicate that your applications are ready.

In order for applications to benefit, the people who write the applications and configure the infrastructure need to support this method of working. Without people ready to give up control to software, you'll never realize the benefits.

1 Applications failing in unexpected ways are an exception, but after the failure is recovered, the software should be patched to not require human intervention the next time.

People

People are the hardest part of cloud native infrastructure.

If you want to build an architecture that replaces people's functions and decisions with software, you need to make sure they know you have their best interests in mind. They need to not only accept the change but also be asking for it and building it themselves.

Developing applications is hard; operating infrastructure is hard. Application developers often believe they can replace infrastructure operators with tooling and automation. Infrastructure operators wish application developers would write more reliable code with automatable debugging and recovery. These tensions are the basis for DevOps, which has many other books written about it, including *Effective DevOps*, by Jennifer Davis and Katherine Daniels (O'Reilly, 2016).

People don't scale, nor are they good at repetitive, mundane tasks.

The goal of application and systems engineers should be to take away the mundane and repetitive tasks so they can focus on more interesting problems. They need to have the skills to develop software that can contain their business logic and decisions. There needs to be enough engineers to write the needed software, and more importantly, maintain it.

The most critical aspect is that they need to work together. One side of engineering cannot migrate to a new way of running and managing applications without the support of the other. Team organization and communication structure is important.

We will address team readiness soon, but first we must decide when infrastructure systems are ready for cloud native infrastructure.

Systems

Cloud native applications need system abstractions. The application should not be concerned with an individual, hardcoded hostname. If your applications cannot run without careful placement on individual hosts, then your systems are not ready for cloud native infrastructure.

Taking a single server (virtual or physical) running an operating system and turning it into a method by which to access resources is what we mean when we say "abstractions." Individual systems should not be the target of deployment for an application. Resources (CPU, RAM, and disk) should be pooled across all available machines and then divvied up by the platform from applications' requests.

In cloud native infrastructure, you must hide underlying systems to improve reliability. Cloud infrastructure, like applications, expects failures of underlying components

to occur and is designed to handle such failures gracefully. This is needed because the infrastructure engineers no longer have control of everything in the stack.

Is Kubernetes Cloud Native Infrastructure?

Kubernetes is a framework that makes managing applications easier and promotes doing so in a cloud native way. However, you can also use Kubernetes in a very un-cloud-native way.

Kubernetes exposes extensions to build on top of its core functionality, but it is not the end goal for your infrastructure. Other projects (e.g., OpenShift) build on top of it to abstract away Kubernetes from the developer and applications.

Platforms are where your applications should run. Cloud native infrastructure supports them but also encourages ways to run infrastructure.

If your applications are dynamic but your infrastructure is static, you will soon reach an impasse that cannot be addressed with Kubernetes alone.

Infrastructure is ready to become cloud native when it is no longer a challenge. Once infrastructure becomes easy, automated, self-serviceable, and dynamic, it has the potential to be ignored. When systems can be ignored and the technology becomes mundane, it's time to move up the stack.

If your systems management relies on ordering hardware or running in a "hybrid cloud," your systems may not be ready. It is possible to manage a data center and be cloud native. You will need to be vigilant in separating the responsibilities of building data centers from those of managing infrastructure.

Google, Facebook, Amazon, and Microsoft have all found benefits in creating hardware from scratch through their Open Compute Project. Needing to create their own hardware was a limitation of performance and cost. Because there is a clear separation of responsibilities between hardware design and infrastructure builders, these companies are able to run cloud native infrastructure at the same time they're creating custom hardware. They are not hindered by running "on-prem." Instead they can optimize their hardware and software together to gain more efficiency and performance out of their infrastructure.

Managing your own data center is a big investment of time and money. Creating an on-premises cloud is too. Doing both will require teams to build and manage the data center, teams to create and maintain the APIs, and teams to create abstractions on top of the IaaS APIs.

All of this can be done, and it is up to your business to decide if there is value in managing the entire stack.

Now we can look at what other areas of business need to be prepared for a shift to cloud native practices.

Business

> If the architecture of the system and the architecture of the organization are at odds, the architecture of the organization wins.
>
> —Ruth Malan, "Conway's Law"

Businesses are very slow to change. They may be ready to adopt cloud native practices when scaling people to manage scaling systems is no longer working, and when product development requires more agility.

People don't scale infinitely. For each person added to manage more servers or develop more code, there is a strain on the human infrastructure that supports them (e.g., office space). There is also overhead for other people because there needs to be more communication and more coordination.

As we discussed in Chapter 1, by using a public cloud, you can reduce some of the process and people overhead by renting server time. Even with a public cloud, you will still have people that manage infrastructure details (e.g., servers, services, and user accounts).

The business is ready to adopt cloud native practices when the communication structure reflects the infrastructure and applications the business needs to create. This includes communications structures that mirror architectures like microservices. They could be small, independent teams that do not have to go through layers of management to talk to or work with other teams.

DevOps and Cloud Native

DevOps can complement the way teams work together and influence the type of tooling used. It has many benefits for companies that adopt it, including rapid prototyping and increased deployment velocity. It also has a strong focus on the culture of the organization.

Cloud native needs high-performing organizations, but focuses on design, architecture, and hygiene more than team workflows and culture. However, if you think you can implement successful cloud native patterns without addressing how your application developers, infrastructure operators, and anyone in the technology department interact, you may be in for a surprise.

Another limiting factor that forces business change is needing more application agility. Businesses need to not only deploy fast, but also drastically change what they deploy.

The raw number of deploys does not matter. What matters is providing customer value as quickly as possible. Believing the software deployed will meet all of the customer's needs the first time, or even the 100th time, is a fallacy.

When the business realizes it needs to iterate and change frequently, it may be ready to adopt cloud native applications. As soon as it finds limitations in people efficiency and old process limitations, and it's open to change, it is ready for cloud native infrastructure.

All the factors that indicate when to adopt cloud native don't tell the full story. Any design is about trade-offs. So here are some situations when cloud native infrastructure is not the right choice.

When You Don't Need Cloud Native Infrastructure

Understanding the benefits of a system is important only when you know the limitations. That is, knowing which limitations are acceptable for your needs can often be more of a deciding factor than the benefits.

It is also important to keep in mind that needs change over time. The functionality that is critical to have now may not be in the future. Likewise, if the following situations don't make this architecture ideal now, you have control over many of these situations and can change to adopt them.

Technical Limitations

Just like applications, with infrastructure the easiest items to change are technical. If you know when you should adopt cloud native infrastructure based on technical merit, you can reverse those traits to also find out when you should *not* adopt cloud native infrastructure.

The first of these limitations is not having cloud native applications. As discussed in Chapter 1, if your applications need human interaction, whether it be scheduling, restarting, or searching logs, cloud native infrastructure will be of little benefit.

Even having an application that can be dynamically scheduled does not make it cloud native. If your application runs on Kubernetes but still requires humans to set up monitoring, log collection, or a load balancer, it's not cloud native. And just because you have Kubernetes does not mean you're done.

If you have a orchestrator, it's important to look at what it took to get it running. Did you have to place a purchase order, create a ticket, or send an email to get servers?

Those are indicators you don't have self-service infrastructure, which is a requirement for a cloud.

In the cloud you provide nothing more than billing information and call an API. Even if you run servers on-prem, you should have a team that builds IaaS, and then you layer the cloud native infrastructure on top.

If you're building a cloud environment in your own data center, Figure 2-1 shows an example of where your underlying infrastructure components fit. All raw components (e.g., compute, storage, network) should be available from a self-service IaaS API.

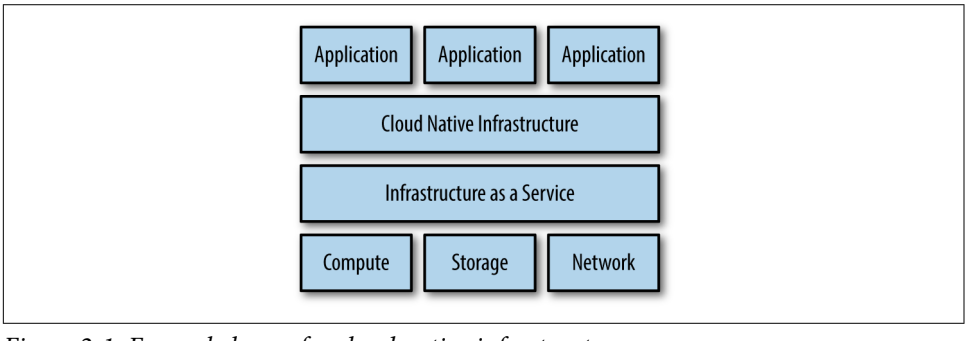


Figure 2-1. Example layers for cloud native infrastructure

In a public cloud, you have IaaS and hosted services, but that doesn't mean your business is ready for what the public cloud can enable.

When you're building a platform to run applications, it's important to know what you are getting into. Initial development is only a small fraction of what it takes to build and maintain a platform, especially one that is business critical.

Maintenance typically consumes about 40 to 80 percent (60 percent on average) of software costs. Therefore, it is probably the most important life cycle phase.[2]

Discovering the business requirements and building the skills needed to do the development might be too much for a small team. Once you have the skills to develop the needed platform, you still need to invest time to improve and maintain the system. This takes considerably longer than initial development.

2 Robert L. Glass, "Frequently Forgotten Fundamental Facts about Software Engineering," *IEEE Software*, vol. 18, no. 3 (May/June 2001): 110–111.

Providing the absolute best environment for businesses to run is the product of public cloud providers. If you're not able or willing to also make your platform a differentiator for your business, then you should not build one yourself.

Keep in mind that you don't have to build everything yourself. You can use services or products that can be assembled into the platform you need.

Reliability is still a key feature of your infrastructure. If you're not ready to give up control to lower levels of the infrastructure stack and still make a reliable product through embracing failures, then cloud native infrastructure is not the right choice.

There are also limitations that may be out of your control. The nontechnical limitations are just as important and harder to fix.

Business Limitations

If existing processes don't support changing the infrastructure, then you need to surmount that obstacle first. Luckily you do not have to do it alone.

This book will hopefully help clearly explain the benefits and process to people who need convincing. There are also many case studies and companies sharing their experience of adopting these practices. You will find one case study in this book, but it is up to you to find relevant examples for your business and to share them with your peers and management.

If the business doesn't already have an avenue for experimentation and a culture that supports trying new things (and the consequences that come with failure), then changing processes will likely be impossible. In this case, your options are to reach a critical point where things have to change or to convince management that change is necessary.

It is impossible to tell from the outside if a business is ready to adopt cloud native architecture. Some processes that are clear indicators that the company is not ready include:

- Resource requests that require human intervention
- Regularly scheduled maintenance windows that require manual labor
- Manual inventory tracking and resource assignments
- Spreadsheet inventory

If people besides the team responsible for a service are involved in the process to schedule, deploy, upgrade, or monitor the service, you may need to address those processes before or during a migration to cloud native infrastructure.

There are also sometimes processes outside of the business's control, such as industry regulations. Sadly, these are even harder and slower to change than internal processes.

If industry regulations limit the velocity or agility of development, we have no advice, except do what you can. If regulations don't allow running in a public cloud, try your best to use technologies to run one on-premises. Management will need to make a case for changing regulations to whatever governing body sets them.

There is one other nontechnical hindrance to cloud native infrastructure. In some companies, there is a culture of not using third-party services.[3]

If your company is not willing, or via process not able, to use third-party, hosted services, it may not be the right time to adopt cloud native infrastructure.

Conclusion

> To succeed, planning alone is insufficient. One must improvise as well.
>
> —Isaac Asimov

Throughout this chapter we have discussed considerations of when to adopt cloud native infrastructure. There are many areas to keep in mind and each situation is unique. Hopefully, some of these guidelines can help you discover the opportune time to make the change.

If your company has already been adopting some cloud native practices, these questions can help identify other areas that can also adopt this architecture. It is important to know if the trade-offs and benefits are the right solution, when you should adopt them, and how you can get started.

If you are not already applying cloud native practices at your work, there are no shortcuts. The business and employees will need to collectively decide it is the right solution and make progress together. No one succeeds alone.

3 Often called "Not Invented Here" syndrome.

CHAPTER 3
Evolution of Cloud Native Deployments

We discussed in the previous chapter what the requirements are before you adopt cloud native infrastructure. Having API-driven infrastructure provisioning (IaaS) will be a requirement before you deploy.

In this chapter, we explore taking the idea of cloud native infrastructure topologies and realizing them in a cloud. We'll learn common tools and patterns that help operators control their infrastructure.

The first step in deploying infrastructure is being able to represent it. Traditionally this could have been handled on a whiteboard, or, if you're lucky, in documentation stored on a company wiki. Today, things are moving into a more programmatic space, and infrastructure representation is commonly documented in ways that make it easy for applications to interpret it. Regardless of the avenue in which it is represented, the need to comprehensively represent the infrastructure remains the same.

As one would expect, defining infrastructure in the cloud can range from very simple designs to very complex designs. Regardless of the complexity, great attention to detail must be given to infrastructure representation to ensure the design is reproducible. Being able to clearly transmit ideas is even more important. Therefore, having a clear, accurate, and easy-to-understand representation of infrastructure-level resources is imperative.

We also gain a lot from having a well-crafted representation:

- Infrastructure design can be shared and versioned over time.
- Infrastructure design can be forked and altered for unique situations.
- The representation implicitly is documentation.

As we move forward in the chapter, we will see how infrastructure representation is the first step in infrastructure deployment. We will explore both the power and potential pitfalls of representing infrastructure in different ways.

Representing Infrastructure

To begin, we need to understand the two roles in representing infrastructure: the author and the audience.

The *author* is what will be defining the infrastructure, usually a human operator or administrator. The *audience* is what will be responsible for interpreting the infrastructure representation. Sometimes this is a human operator performing manual steps, but hopefully it is a deployment tool that can automatically parse and create the infrastructure. The better the author is at accurately representing the infrastructure, the more confidence we can gain in the audience's ability to interpret the representation.

The main concern while authoring infrastructure representation is to make it understandable for the audience. If the target audience is humans, the representation might come in the form of a technical diagram or abstracted code. If the target audience is a program, the representation may need more detailed information and concrete implementation steps.

Despite the audience, the author should make it easy for the audience to consume. This becomes difficult as complexity increases and if the representation is consumed by both humans and programs.

Representation needs to be easy to understand so that it can be accurately parsed. Easy-to-read but inaccurately parsed representation negates the entire effort. The audience should always strive to interpret the representation they were given, and never make assumptions.

In order to make the representation successful, the interpretation needs to be predictable. The best audiences are ones that will fail quickly and obviously if the author neglected an important detail. Having predictability will reduce mistakes and errors when changes are applied, and will help build trust between authors and audiences.

Infrastructure as a Diagram

We all have walked up to a whiteboard and began drawing an infrastructure diagram. Usually, this starts off with a cloud shape in the corner that represents the internet, and some arrows pointing to boxes. Each of these boxes represents a component of the system, and the arrows represent the interactions between them. Figure 3-1 is an example of an infrastructure diagram.

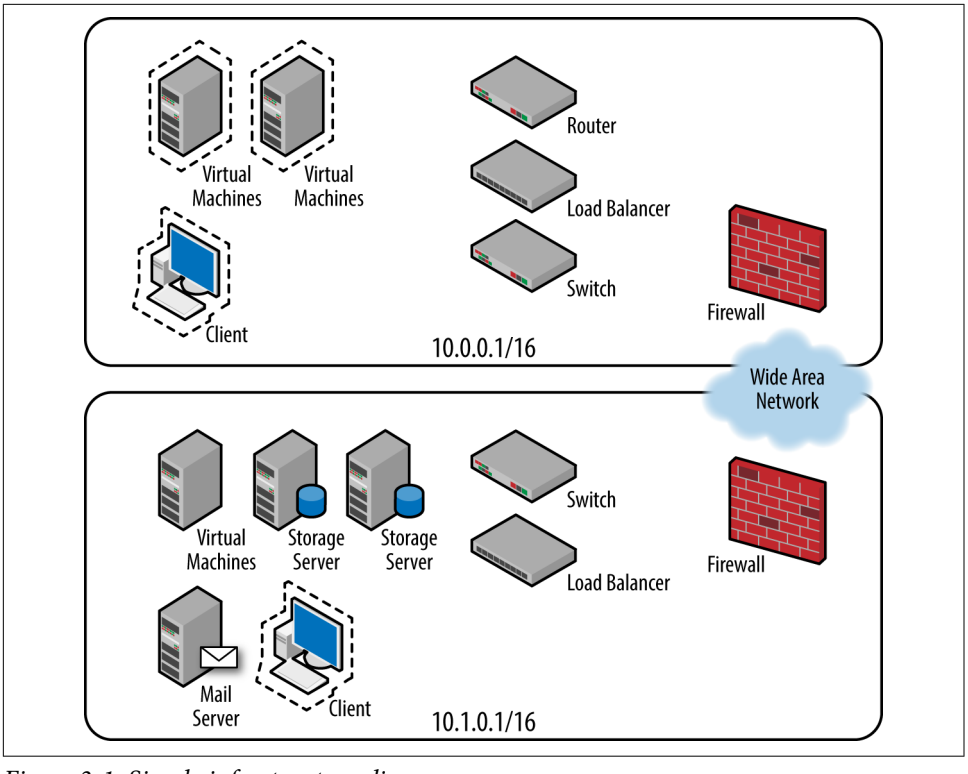


Figure 3-1. Simple infrastructure diagram

This is a wonderfully effective approach to brainstorming and transmitting ideas to other humans. It allows for quick and powerful representation of complex infrastructure designs.

Pictures work well for humans, large crowds, and CEOs. These diagrams also work well because they use common idioms to represent relationships. For example, this box may send data to that box, but will never send data to this other box.

Unfortunately, diagrams are virtually impossible for a computer to understand. Until computer vision catches up, infrastructure diagrams remain a representation to be interpreted by eyeballs, not code.

Deploying from a Diagram

In Example 3-1 we look at a familiar snippet of code from a `bash_history` file. It represents an infrastructure operator working as the audience for a diagram describing a basic server with networking, storage, and kublet service running.

The operator has manually deployed a new virtual machine, and has SSHed into the machine to begin provisioning it. In this case, the human acts as the interpreter for the diagram and then takes action in the infrastructure environment.

Most infrastructure engineers have done this in their career, and these steps should be alarmingly familiar to some system administrators.

Example 3-1. bash_history

```
sudo emacs /etc/networking/interfaces
sudo ifdown eth0
sudo ifup eth0
sudo fdisk -l
sudo emacs /etc/fstab
sudo mount -a
sudo systemctl enable kubelet
```

Infrastructure as a Script

As a systems administrator, part of your job is to make changes across complex systems; it is also your responsibility to ensure that those changes are correct. The need to have these changes spread across vast systems is very real. Unfortunately, so is human error. It's no surprise that administrators write convenience scripts for the job.

Scripts can help reduce the amount of human error on repeated tasks, but automation is a double-edged sword. It does not imply accuracy or success.

> For SRE, automation is a force multiplier, not a panacea. Of course, just multiplying force does not naturally change the accuracy of where that force is applied: doing automation thoughtlessly can create as many problems as it solves.
>
> —Niall Murphy with John Looney and Michael Kacirek, *The Evolution of Automation at Google*

Writing scripts is a good way to automate steps to produce a desired outcome. The script could perform miscellaneous tasks, such as installing an HTTP server, configuring it, and running it. However, the steps in the scripts rarely take into consideration their outcome or the state of the system when they're invoked.

The script, in this case, is the encoded data that represents what should happen to create the desired infrastructure. Another operator or administrator could evaluate your script and hope to understand what the script is doing. In other words, they'd be

interpreting your infrastructure representation. Understanding the desired infrastructure relies on understanding how the steps affect the system.

The runtime for your script would execute the steps in the order they are defined, but the runtime has no knowledge of what it is producing. The script is the code, and the outcome of the script is, hopefully, the desired infrastructure.

This works well for very basic scenarios, but there are some flaws in this approach. The most obvious flaw would be running the same script and getting two outcomes.

What if the environment where the script was first run is drastically different than the environment where it is run a second time? Scientifically speaking, this would be analogous to flaw in a procedure, and would invalidate the data from the experiment.

Another flaw with using scripts to represent infrastructure is the lack of declarative state. The runtime for the script doesn't understand the end state because it is only provided steps to perform. Humans need to interpret the desired outcome from the steps to understand how to make changes.

We have all seen code that is hard to understand as a human. As the complexity of a provisioning script grows, our ability to interpret what the script does diminishes. Furthermore, as your infrastructure needs change over time, the script will inevitably need to change.

Without abstracting steps into declarative state, the script will grow to try to create procedures for every possible initial state. This includes abstracting away steps and differences between operating systems (e.g., `apt` and `dnf`) as well as verifying what steps can safely be skipped.

Infrastructure as code brought about tools that provided some of these abstractions to help reduce the burden and maintenance of managing infrastructure as scripts.

Deploying from a Script

The next evolution of creating infrastructure is to begin to take the previous process of manually managing infrastructure and simplify it for the administrator by encapsulating the work in a script. Imagine we had a `bash` script called *createVm.sh* that would create a virtual machine from our local workstation.

The script would take two arguments. The first would be the static IP address to assign to a network interface on the virtual machine. The second would be a size in gigabytes that would be used to create a volume and attach it to the virtual machine.

Example 3-2 shows a very basic representation of infrastructure as a script. The script will provision the new infrastructure and run an arbitrary provision script on the

newly created infrastructure. This script could be evolved to be highly customizable and could (dangerously) be automated to run at the click of a button.

Example 3-2. Infrastructure as a script

```
#!/bin/bash
# Create a VM with a NIC on 10.0.0.17 and a 100gb volume
createVm.sh 10.0.0.17 100
# Transfer the bootstrapping script
scp ~/vm_provision.sh user@10.0.0.17:vm_provision.sh -v
# Run the bootstrap script
ssh user@10.0.0.17 sh ~/vm_provision.sh
```

Infrastructure as Code

Configuration management was once the king of representing infrastructure.[1] We can think of configuration management as abstracted scripts that automatically consider initial state to perform the proper procedures. Most importantly, configuration management allows authors to declare the desired state of a node instead of every step needed to achieve it.

Configuration management was the first step in the direction of infrastructure as code, but the tooling has rarely reached beyond the scope of a single server. Configuration management tools do an excellent job of defining specific resources and what their state should be, but complications arise as infrastructure requires coordination between resources.

For example, the DNS entry for a service should not be available until the service has been provisioned. The service should not be provisioned before the host is available. Failure to coordinate multiple resources across independent nodes has made the abstractions provided by configuration management inadequate. Some tools have added the ability to coordinate configuration between resources, but the coordination has often been procedural, and the responsibility has fallen on humans to order resources and understand desired state.[2]

Your infrastructure does not consist of independent entities without communication. Your tools to represent the infrastructure need to take that into consideration. So another representation was needed to manage low-level abstractions (e.g., operating systems), as well as provisioning and coordination.

In July of 2014 an open source tool that embraced the idea of a higher level of abstraction for infrastructure as code was released. The tool, called Terraform, has

1 For certain layers of infrastructure (e.g., OS automation), it still is.

2 Similar to scripts in the previous section.

been fantastically successful. It was released at the perfect time, when configuration management was well established and public cloud adoption was on the rise. Users began to see the limitations of the tools for the new environment, and Terraform was ready to address their needs.

> We originally thought of infrastructure as code in 2011. We noticed we were writing tools to solve the infrastructure problem for many projects, and wanted to standardize the process.
>
> —Mitchell Hashimoto, CEO of Hashicorp and creator of Terraform

Terraform represents infrastructure using a specialized domain-specific language (DSL), which provides a good compromise between human-understandable images and machine-parsible code. Some of the most successful parts of Terraform are its abstracted view of infrastructure, resource coordination, and ability to leverage existing tools when applicable. Terraform talks to cloud APIs to provision infrastructure and can use configuration management to provision nodes when necessary.

This was a fundamental shift in the industry, as we saw one-off provisioning scripts fade into the background. More and more operators began developing infrastructure representation in the new DSL. Engineers who used to manually operate on infrastructure were now developing code.

The new DSL solved the problems of representing infrastructure as a script, and became a standard for representing infrastructure. Engineers found themselves developing a better representation of infrastructure as code, and allowing Terraform to interpret it. Just as with configuration management code, engineers began storing their infrastructure representations in version control and treating infrastructure architecture as they treat software.

By having a standardized way of representing infrastructure, we liberated ourselves from the pain of having to learn every proprietary cloud API. While not all cloud resources could be abstracted in a single representation, most users could accept cloud lock-in in their code. Having a human-readable and machine-parsible representation of infrastructure architecture, not just independent resource declaration, changed the industry forever.

Deploying from Code

After running into the challenges of deploying infrastructure as a script, we have created a program that will parse input and take action against infrastructure for us.

Example 3-3 shows a Terraform configuration taken from the Terraform open source repository. Notice how the code has variables and will need to be parsed at runtime.

Declarative representation of infrastructure is important because it doesn't define individual steps to create the infrastructure. This allows us to separate *what* needs to be provisioned from *how* it gets provisioned. This is what makes this infrastructure representation a new paradigm; it was also a first step in the evolution toward infrastructure as software.

Representing infrastructure in this way is a powerful, common practice for engineers. The user could use Terraform to `terraform apply` the infrastructure.

Example 3-3. example.tf

```
# Create our DNSimple record
resource "dnsimple_record" "web" {
  domain = "${var.dnsimple_domain}"
  name = "terraform"
  value = "${hostname}"
  type = "CNAME"
  ttl  = 3600
}
```

Infrastructure as Software

Infrastructure as code was a powerful move in the right direction. But code is a static representation of infrastructure and has limitations. You can automate the process of deploying code changes, but unless the deployment tool runs continually, there will still be configuration drift. Deployment tooling traditionally works only in a single direction: it can only create new objects, and can't easily delete or modify existing objects.

To master infrastructure, our deployment tools need to work from the initial representation of infrastructure, and mutate the data to make more agile systems. As we begin to look at our infrastructure representation as a versionable body of data that continually enforces the desired state, we need the next step of infrastructure as software.

Terraform took lessons from configuration management and improved on that concept to better provision infrastructure and coordinate resources. Applications need an abstraction layer to utilize resources more efficiently. As we explained in Chap-

ter 1, applications cannot run directly on IaaS and need to run on a platform that can manage resources and run applications.

IaaS presented raw components as provisionable API endpoints, and platforms present APIs for resources that are more easily consumed by applications. Some of those resources may provision IaaS components (e.g., load balancers or disk volumes), but many of them will be be managed by the platform (e.g., compute resources).

Platforms expose a new layer of infrastructure and continually enforce the desired state. The components of the platforms are also applications themselves that can be managed with the same desired state declarations.

The API machinery allows users to reap the benefits of standardizing infrastructure as code, and adds the ability to version and change the representation over time. APIs allow a new way of consuming resources through standard practices such as API versioning. Consumers of the API can build their applications to a specific version and trust that their usage will not break until they choose to consume a new API version. Some of these practices are critical features missing from previous infrastructure as code tools.

With software that continually enforces representation, we can now guarantee the current state of our systems. The platform layer becomes much more consumable for applications by providing the right abstractions.

You may be drawing parallels between the evolution of infrastructure and the evolution of software. The two layers in the stack have evolved in remarkably similar ways.

> Software is eating the world.
>
> —Marc Andreessen

Encapsulating infrastructure and thinking of it as a versioned API is remarkably powerful. This dramatically increases velocity of a software project responsible for interpreting a representation. Abstractions provided by a platform are necessary to keep up with the quickly growing cloud. This new pattern is the pattern of today, and the one that has been proven to scale to unfathomable numbers for both infrastructure and applications.

Deploying from Software

A fundamental difference between infrastructure as code and infrastructure as software is that software has the ability to mutate the data store, and thus the representation of infrastructure. It is up to the software to manage the infrastructure, and the representation is a give-and-take between the operator and the software.

In Example 3-4 we take a look at a YAML representation of infrastructure. We can trust the software to interpret this representation and negotiate the outcome of the YAML for us.

We start with a representation of the infrastructure just as we did when developing infrastructure as code. But in this example the software will run continually and ensure this representation over time. In a sense, this is still read only, but the software can extend this definition to add its own meta information, such as tagging and resource creation times.

Example 3-4. infrastructure.yaml

```
location: "New York 1"
name: example
dns:
    fqdn: infra.example.com
network:
  cidr: 172.0.0.0/12
serverPools:
  - bootstrapScript: /home/user/bootstrap.sh
    diskSize: 40gb
    firewalls:
      - rules:
          - ingressFromPort: 443
            ingressProtocol: tcp
            ingressSource: 0.0.0.0/0
            ingressToPort: 443
    maxCount: 1
    minCount: 1
    image: centos-amd64-7
    subnets:
      - cidr: 172.0.100.0/24
```

Deployment Tools

We now understand the two roles in deploying infrastructure:

The author
: The component defining the infrastructure

The audience
: The deployment tool interpreting the representation and taking action

There are many avenues in which we might represent infrastructure, and the component taking action is a logical reflection of the initial representation. It's important to accurately represent the proper layer of the infrastructure and to eliminate complexity in that layer as much as possible. With simple, targeted releases, we will be able to more accurately apply the changes needed.

Site Reliability Engineering (O'Reilly, 2016) concludes that "simple releases are generally better than complicated releases. It is much easier to measure and understand the impact of a single change rather than a batch of changes released simultaneously."

As our representation of infrastructure has changed over time to be more abstracted from underlying components, our deployment tools have changed to match the new abstraction targets.

We are approaching the infrastructure as software boundary and can notice the early signs of a new era of infrastructure deployment tools. Open source projects across the internet are popping up that claim to be able to manage infrastructure over time. It is the engineer's job to know what layer of infrastructure that project manages and how it impacts their existing tools and other infrastructure layers.

The first step in the direction of cloud native infrastructure was taking provisioning scripts and scheduling them to run continually. Some engineers would intentionally engineer these scripts to work well with being scheduled over time. We began to see elaborate global locking mechanisms, advanced scheduling tactics, and distributed scheduling approaches.

This was essentially what configuration management promised, albeit at a more resource-specific abstraction. Thanks to the cloud, the days of automating scripts to manage infrastructure have come and gone.

> Automation is dead.
>
> —Charity Majors, CEO of Honeycomb

We are imagining a world where we begin to look at infrastructure tooling completely differently. If your infrastructure is designed to run in the cloud, then IaaS is not the problem you should be solving. Consume the provided APIs, and build the new layer of infrastructure that can be directly consumed by applications.[3]

We are in a special place in the evolution of infrastructure where we take the leap into designing infrastructure deployment tools as elegant applications from day one.

Good deployment tools are tools that can quickly go from a human-friendly representation of infrastructure to working infrastructure. Better deployment tools are tools that will undo any changes that have been made that disagree with the initial representation. The best deployment tools do all of this without needing human involvement.

As we build these applications, we cannot forget important lessons learned from provisioning tools and software practices that are crucial for handling complex systems.

3 If you are able, don't build the layer of infrastructure at all.

Some key aspects of deployment tools we'll look at are idempotency and handling failure.

Idempotency

Software can be *idempotent*, meaning you must be able to continually feed it the same input, and always get the same output.

In technology, this idea was made famous by the Hypertext Transfer Protocol (HTTP) via idempotent methods like `PUT` and `DELETE`. This is a very powerful idea, and advertising the guarantee of idempotency in software can drastically shape complex software applications.

One of the lessons learned in early configuration management tools was idempotency. We need to remember the value this feature offered infrastructure engineers, and continue to build this paradigm into our tooling.

Being able to automatically create, update, or delete infrastructure with the guarantee that no matter how often you run the task it will always output the same is quite exciting. It allows for operators to begin to work at automating tasks and chores. What used to be a sizable amount of work for an operator can now be as simple as a button click in a web page.

The idempotent guarantee is also effective at helping operators perform quality science on their infrastructure. Operators could begin replicating infrastructure in many physical locations, with the knowledge that someone else repeating their procedure would get the same thing.

We began to notice entire frameworks and toolchains built around this idea of automating arbitrary tasks for repeatability.

As it was with software, so it became with infrastructure. Operators began automating entire pipelines of managing infrastructure using these representations and deployment tools. The work of an operator now became developing the tooling around automating these tasks, and no longer performing the tasks themselves.

Handling Failure

Any software engineer can tell you about the importance of handling failures and edge cases in their code. We naturally began to develop these same concerns as infrastructure administrators.

What happens if a deployment job fails in the middle of its execution, or, more importantly, what *should* happen in that case?

Designing our deployment tools with failure in mind was another step in the right direction. We found ourselves sending messages or registering alerts in a monitoring

system. We kept detailed logs of the automation tasks. We even made the leap to chaining logic together in the case of failure.

We obsessed over failures. We began taking action in the case of failures and took action when they happened.

But building systems around the idea that a single component might fail is drastically different than building components to be more resilient to failure. Having a component retry or adjust its approach based on failure is taking the resiliency of a system a step deeper into the software. This allows for a much stabler system and reduces the overall support needed from the system itself.

Design components for failure, not systems.

Eventual consistency

In the name of designing components for failure, we need to learn a term that describes a common approach at handling failure.

Eventual consistency means to attempt to reconcile a system over time. Larger systems and smaller components can both adhere to this philosophy of retrying a failed process over time.

One of the benefits of an eventually consistent system is that an operator can have confidence that it will ultimately reach a desired state. One of the concerns with these systems is that sometimes they might take inappropriately long amounts of time to reach the desired state.

Knowing when to choose a stable but slow system versus an unreliable but fast system is a technical decision an administrator must make. The important relationship to note in this decision is that systems exchange speed for reliability. It's not always easy, but if in doubt, always choose reliable systems.

Atomicity

Contrary to eventually consistent systems is the *atomic system*, a guaranteed transaction that dictates that the job succeeded in its entirety. If the job cannot complete, it will revert any changes it made and fail completely.

Imagine a job that needed to create 10 virtual machines. The job gets to the 7th virtual machine and something goes wrong. According the eventual consistency approach, we would just try the job over and over in the hopes of eventually getting 10 virtual machines.

It's important to look at the reason we only were able to create seven virtual machines. Imagine there was a limit on how many virtual machines a cloud would let us create. The eventual consistency model would continue to try to create three more machines and inevitably fail every time.

If the job were engineered to be atomic, it would hit the limit on the seventh machine and realize there was a catastrophic failure. The job would then be responsible for deleting the partial system.

So the operator can rest assured that they either get the system as intended in its entirety or nothing at all. This is a powerful idea, as many components in infrastructure are dependent on the rest of the system to be in place in order for it to work.

We can introduce confidence in exchange for inconvenience. That is to say, the administrator would be confident the state of their system would never change unless a perfect change could be applied. In exchange for this perfect system, the operator might face great inconvenience, as the system could require a lot of work to keep running smoothly.

Choosing an atomic system is safe, but potentially not what we want. The engineer needs to know what system they want and when to pick atomicity versus eventual consistency.

Conclusion

The pattern of deploying infrastructure is simple and has remained unchanged since before the cloud was available. We represent infrastructure, and then using some device, we manifest the infrastructure into reality.

The infrastructure layers of the stack have astonishingly similar histories to the software application layers. Cloud native infrastructure is no different. We begin to find ourselves repeating history, and learning age-old lessons in new guises.

What is to be said about the ability to predict the future of the infrastructure industry if we already know the future of its software counterpart?

Cloud native infrastructure is a natural, and possibly expected, evolution of infrastructure. Being able to deploy, represent, and manage it in reliable and repeatable ways is a necessity. Being able to evolve our deployment tools over time, and shift our paradigms of how this is done, is critical to keeping our infrastructure in a space that can keep up with supporting its application-layer counterpart.

CHAPTER 4
Designing Infrastructure Applications

In the previous chapter we learned about representing infrastructure and the various approaches and concerns with deployment tools around it. In this chapter we look at what it takes to design applications that deploy and manage infrastructure. We heed the concerns of the previous chapter and focus on opening up the world of *infrastructure as software*, sometimes called *infrastructure as an application*.

In a cloud native environment, traditional infrastructure operators need to be infrastructure software engineers. It is still an emerging practice and differs from other operational roles in the past. We desperately need to begin exploring patterns and setting standards.

A fundamental difference between infrastructure as code and infrastructure as software is that software continually runs and will create or mutate infrastructure based on the reconciler pattern, which we will explain later in this chapter. Furthermore, the new paradigm behind infrastructure as software is that the software now has a more traditional relationship with the data store and exposes an API for defining desired state. For instance, the software might mutate the representation of infrastructure as needed in the data store, and very well could manage the data store itself! Desired state changes to reconcile are sent to the software via the API instead of static code repo.

The first step in the direction of infrastructure as software is for infrastructure operators to realize they are software engineers. We welcome you all warmly to the field! Previous tools (e.g., configuration management) had similar goals to change infrastructure operators' job function, but often the operators only learned how to write a limited DSL with narrow scope application (i.e., single node abstraction).

As an infrastructure engineer, you are tasked not only with having a mastery of the underlying principals of designing, managing, and operating infrastructure, but also

with taking your expertise and encapsulating it in the form of a rock-solid application. These applications represent the infrastructure that we will be managing and mutating.

Engineering software to manage infrastructure is not an easy undertaking. We have all the major problems and concerns of a traditional application, and we are developing in an awkward space. It's awkward in the sense that infrastructure engineering is an almost ridiculous task of building software to deploy infrastructure so that you can then run the same software on top of the newly created infrastructure.

To begin, we need to understand the nuances of engineering software in this new space. We will look at patterns proven in the cloud native community to understand the importance of writing clean and logical code in our applications. But first, where does infrastructure come from?

The Bootstrapping Problem

On Sunday, March 22, 1987, Richard M. Stallman sent an email to the GCC mailing list to report successfully compiling the C compiler with itself:

> This compiler compiles itself correctly on the 68020 and did so recently on the vax. It recently compiled Emacs correctly on the 68020, and has also compiled tex-in-C and Kyoto Common Lisp. However, it probably still has numerous bugs that I hope you will find for me.
>
> I will be away for a month, so bugs reported now will not be handled until then.
>
> —Richard M. Stallman

This was a critical turning point in the history of software, as we were engineering software to bootstrap itself. Stallman had literally created a compiler that could compile itself. Even accepting this statement as truth can be philosophically difficult.

Today we are solving the same problem with infrastructure. Engineers must come up with solutions to almost impossible problems of a system bootstrapping itself and coming to life at runtime.

One approach is to provision the first bit of infrastructure in the cloud and infrastructure applications manually. While this approach does work, it usually comes with the caveat that the operator should destroy the initial bootstrap infrastructure after more appropriate infrastructure has been deployed. This approach is tedious, difficult to repeat, and prone to human errors.

A more elegant and cloud native approach to solving this problem is to make the (usually correct) assumption that whoever is attempting to bootstrap infrastructure software has a local machine that we can use to our advantage. The existing machine (your computer) serves as the first deployment tool, to create infrastructure in a cloud automatically. After the infrastructure is in place, your local deployment tool

can then deploy itself to the newly created infrastructure and continually run. Good deployment tools will allow you to easily clean this up when you are done.

After the initial infrastructure bootstrap problem is solved, we can then use the infrastructure applications to bootstrap new infrastructure. The local computer is now taken out of the equation, and we are running entirely cloud native at this point.

The API

In earlier chapters we discussed the various methods for representing infrastructure. In this chapter we will be exploring the concept of having an API for infrastructure.

When the API is implemented in software, it more than likely will be done via a data structure. So, depending on the programming language you are using, it's safe to think of the API as a class, dictionary, array, object, or struct.

The API will be an arbitrary definition of data values, maybe a handful of strings, a few integers, and a boolean. The API will be encoded and decoded from some sort of encoding standing like JSON or YAML, or might even be stored in a database.

Having a versionable API for a program is a common practice for most software engineers. This allows the program to move, change, and grow over time. Engineers can advertise to support older API versions, and offer backward-compatibility guarantees. In engineering infrastructure as software, using an API is preferred for these reasons.

Finding an API as the interface for infrastructure is one of the many clues that a user will be working with infrastructure as software. Traditionally, infrastructure as code is a direct representation of the infrastructure a user will be managing, whereas an API might be an abstraction on top of the exact underlying resources being managed.[1]

Ultimately, an API is just a data structure that represents infrastructure.

The State of the World

Within the context of an infrastructure as software tool, the world is the infrastructure that we will be managing. Thus, the state of the world is just an audited representation of the world as it exists to our program.

1 Remember from Chapter 1 that cloud native applications need more than just raw infrastructure components, and that the abstractions we expose via the API should be directly consumable by applications as part of a platform.

The state of the world will ultimately make its way back to an in-memory representation of the infrastructure. These in-memory representations should map to the original API used to declare infrastructure. The audited API, or state of the world, typically will needed to be saved.

A *storage medium* (sometimes referred to as a *state store*) can be used to store the freshly audited API. The medium can be any traditional storage system, such as a local filesystem, cloud object storage, or a database. If the data is stored in a filesystem-like store, the tool will most likely encode the data in a logical way so that the data can easily be encoded and decoded at runtime. Common encodings for this include JSON, YAML, and TOML.

As you begin to engineer your program, you might catch yourself wanting to store privileged information with the rest of the data you are storing. This may or may not be best practice, depending on your security requirements and where you plan on storing data.

It is important to remember that storing secrets can be a vulnerability. While you are designing software to control the most fundamental part of the stack, security is critical. So it's usually worth the extra effort to ensure secrets are safe.

Aside from storing meta information about the program and cloud provider credentials, an engineer will also need to store information about infrastructure. It is important to remember that the infrastructure will be represented in some way, ideally one that's easy for the program to decode. It is also important to remember that making changes to a system does not happen instantly, but rather over time.

Having these pieces of data stored and easily accessible is a large part of designing the infrastructure management application. The infrastructure definition alone is quite possibly the most intellectually valuable part of the system. Let's take a look at a basic example to see how this data and the program will work together.

It is important to review Examples 4-1 through 4-4, as they are used as concrete examples for lessons further in the chapter.

A filesystem state store example

Imagine a data store that was simply a directory called *state*. Within the directory, there would be three files:

- *meta_information.yaml*
- *secrets.yaml*
- *infrastructure.yaml*

This simple data store can accurately encapsulate the information needed to be preserved in order to effectively manage infrastructure.

The *secrets.yaml* and *infrastructure.yaml* files store the representation of the infrastructure, and the *meta_information.yaml* file (Example 4-1) stores other important information such as when the infrastructure was last provisioned, who provisioned it, and logging information.

Example 4-1. state/meta_information.yaml

```
lastExecution:
  exitCode: 0
  timestamp: 2017-08-01 15:32:11 +00:00
  user: kris
logFile: /var/log/infra.log
```

The second file, *secrets.yaml*, holds private information, used to authenticate in arbitrary ways throughout the execution of the program (Example 4-2).

Again, storing secrets in this way might be unsafe. We are using *secrets.yaml* merely as an example.

Example 4-2. state/secrets.yaml

```
apiAccessToken: a8233fc28d09a9c27b2e2f
apiSecret: 8a2976744f239eaa9287f83b23309023d
privateKeyPath: ~/.ssh/id_rsa
```

The third file, *infrastructure.yaml*, would contain an encoded representation of the API, including the API version used (Example 4-3). Here can we find infrastructure representation, such as as network and DNS information, firewall rules, and virtual machine definitions.

Example 4-3. state/infrastructure.yaml

```
location: "San Francisco 2"
name: infra1
dns:
    fqdn: infra.example.com
```

```
network:
  cidr: 10.0.0.0/12
serverPools:
  - bootstrapScript: /opt/infra/bootstrap.sh
    diskSize: large
    workload: medium
    memory: medium
    subnetHostsCount: 256
    firewalls:
      - rules:
          - ingressFromPort: 22
            ingressProtocol: tcp
            ingressSource: 0.0.0.0/0
            ingressToPort: 22
    image: ubuntu-16-04-x64
```

The *infrastructure.yaml* file at first might appear to be nothing more than an example of infrastructure as code. But if you look closely, you will see that many of the directives defined are an abstraction on top of the concrete infrastructure. For instance, the `subnetHostsCount` directive is an integer value and defines the intended number of hosts for a subnet. The program will manage sectioning off the larger classless interdomain routing (CIDR) value defined in `network` for the operator. The operator does not declare a subnet, just how many hosts they would like. The software reasons about the rest for the operator.

As the program runs, it might update the API and write the new representation out to the data store (which in this case is simply a file). To continue with our `subnetHostsCount` example, let's say that the program did pick out a subnet CIDR for us. The new data structure might look something like Example 4-4.

Example 4-4. state/infrastructure.yaml

```
location: "San Francisco 2"
name: infra1
dns:
    fqdn: infra.example.com
network:
  cidr: 10.0.0.0/12
serverPools:
  - bootstrapScript: /opt/infra/bootstrap.sh
    diskSize: large
    workload: medium
    memory: medium
    subnetHostsCount: 256
    assignedSubnetCIDR: 10.0.100.0/24
    firewalls:
      - rules:
          - ingressFromPort: 22
            ingressProtocol: tcp
```

```
        ingressSource: 0.0.0.0/0
        ingressToPort: 22
  image: ubuntu-16-04-x64
```

Notice how the program wrote the `assignedSubnetCIDR` directive, not the operator. Also remember how the program updating the API is a sign that a user is interacting with infrastructure as software.

Now remember this is just an example, and does not necessarily advocate for using an abstraction for calculating a subnet CIDR. Different use cases may require different abstractions and implementation in the application. One of the beautiful and powerful things about building infrastructure applications is that users can engineer the software in any way they find necessary to solve their set of problems.

The data store (the *infrastructure.yaml* file) can now be thought of as a traditional data store in the software engineering realm. That is, the program can have full write control over the file.

This introduces risk, but also a great win for the engineer, as we will discover. The infrastructure representation doesn't have to be stored in files on a filesystem. Instead, it can be stored in any data storage such as a traditional database or key/value storage system.

To understand the complexities of how software will handle this new representation of infrastructure, we have to understand the two states in the system—the *expected* state in the form of the API, which is found in the *infrastructure.yaml* file, and the *actual* state that can be observed in reality (or audited), or the state of the world.

In this example, the software hasn't done anything or taken any action yet, and we are at the beginning of the management timeline. Thus, the actual state of the world would be nothing, while the expected state of the world would be whatever is encapsulated in the *infrastructure.yaml* file.

The Reconciler Pattern

The *reconciler pattern* is a software pattern that can be used or expanded upon for managing cloud native infrastructure. The pattern enforces the idea of having two representations of the infrastructure—the first being the actual state of the infrastructure, and the second being the expected state of the infrastructure.

The reconciler pattern will force the engineer to have two independent avenues for getting either of these representations, as well as to implement a solution to reconcile the actual state into the expected state.

The reconciler pattern can be thought of as a set of four methods, and four philosophical rules:

1. Use a data structure for all inputs and outputs.
2. Ensure that the data structure is immutable.
3. Keep the resource map simple.
4. Make the actual state match the expected state.

These are powerful guarantees that a consumer of the pattern can rely on. Furthermore, they liberate the consumer from the implementation details.

Rule 1: Use a Data Structure for All Inputs and Outputs

The methods implementing the reconciler pattern must only accept and return a data structure. The structure must be defined outside the context of the reconciler implementation, but the implementation must be aware of it.

By only accepting a data structure for input and returning one as output, the consumer can reconcile any structure defined in their data store without having to be bothered with how that reconciliation takes place. This also allows the implementations to be changed, modified, or switched at runtime or with different versions of the program.

While we want to adhere to the first rule as often as possible, it's also very important to never tightly couple a data structure and codebase. Always observe best abstraction and separation practices, and never use subsets of the API to pass to/from functions or classes.

Rule 2: Ensure That the Data Structure Is Immutable

Think of a data structure like a contract or guarantee. Within the context of the reconciler pattern, the actual and expected structures are set in memory at runtime. This guarantees that before a reconciliation, the structures are accurate. During the process of reconciling infrastructure, if the structure is changed, a new structure with the same guarantee must be created. A wise infrastructure application will enforce data structure immutability such that even if an engineer attempted to mutate a data structure, it wouldn't work, or the program would error (or maybe even not compile).

The core component of an infrastructure application will be its ability to map a representation to a set of resources. A resource is a single task that will need to be run in order to fulfill the infrastructure requirements. Each of these tasks will be responsible for changing infrastructure in some way.

Basic examples could be deploying a new virtual machine, setting up a new network, or provisioning an existing virtual machine. Each of these units of work will be referred to as a *resource*. Each data structure should map to some number of resources. The application is responsible for reasoning about the structure, and creating the

set of resources. An example of how the API maps to individual resources can be seen in Figure 4-1.

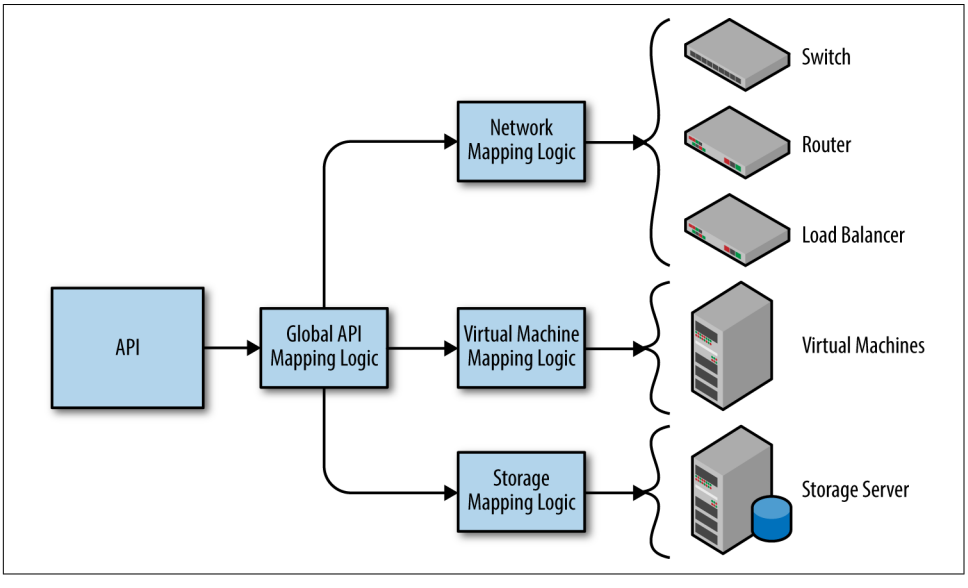


Figure 4-1. Diagram to map a structure to resources

The reconciler pattern demonstrates a stable approach to working with a data structure as it mutates resources. Because the reconciler pattern requires comparing states of resources, it is imperative that data structures be immutable. This dictates that whenever the data structure needs to be updated, a new data structure must be created.

Be mindful of infrastructure mutations. Every time a mutation occurs, the actual data structure is then stale. A clever infrastructure application will be aware of this concern and handle it accordingly.

A simple solution would be to update the data structure in memory whenever a mutation occurs. If the actual state is updated with every mutation, then the reconciliation process can be observed as the actual state going through a set of changes over time until it ultimately matches the expected state and the reconciliation is complete.

Rule 3: Keep the Resource Map Simple

Behind the scenes of the reconciler pattern is an *implementation*. An implementation is just a set of code that has methods to create, modify, and delete infrastructure. A program might have many implementations.

Each implementation will ultimately need to map a data structure to some set of resources. The set of resources will need to be grouped together in a logical way so that the program can reason about each of the resources.

Other than having the basic model of the resources created, you must give great attention to each resource's dependencies. Many resources have dependencies on other resources, meaning that many pieces of infrastructure depend on other pieces to be in place. For example, a network will need to exist before a virtual machine can be placed in the network.

The reconciler pattern dictates that the simplest data structure for grouping resources should be used.

Solving the resource map problem is an engineering decision and might change for each implementation. It is important to pick a data structure carefully, as the reconciler needs to be stable and approachable from an engineering perspective.

Two common structures for mapping data are sets and graphs.

A *set* is a flat list of resources that can be iterated on. In many programming languages, these are called lists, sets, arrays, or the like.

A *graph* is a collection of vertices that are linked together via pointers. The *vertex* of a graph is usually a struct or a class, depending on the programming language. A vertex has a link to another vertex via a pointer defined somewhere in the vertex. A graph implementation can visit each of the vertices by hopping from one to the other via the pointer.

Example 4-5 is an example of a basic vertex in the Go programming language.

Example 4-5. Example vertex

```
// Vertex is a data structure that represents a single point on a graph.
// A single Vertex can have N number of children vertices, or none at all.
type Vertex struct {
    Name string
    Children []*Vertex
}
```

An example of traversing the graph might be as simple as recursively iterating through each of the children. This traversal is sometimes called *walking the graph*.

Example 4-6 is an example of recursively visiting every vertex in the graph via a depth-first traversal written in Go.

Example 4-6. Depth-first traversal

```
// recursiveWalk will recursively dig into all children,
// and their children accordingly and echo the name of
// the vertex currently being visited to STDOUT.
func recursiveWalk(v *Vertex){
    fmt.Printf("Currently visiting vertex: %s\n", v.Name)
    for _, child := range v.Children {
        recursiveWalk(child)
    }
}
```

At first, a simple implementation of a graph seems like a reasonable choice for solving the resource map, as dependencies can be handled by building the graph in a logical way. While a graph would work, it also introduces risk and complexity. The biggest risk with implementing a graph to map resources would be having cycles in the graph. A *cycle* is when one vertex of a graph points to another vertex via more than one path, meaning that traversing the graph is an endless operation.

A graph can be used when necessary, but for most cases, the reconciler pattern should be mapped with a set of resources, not a graph. Using a set allows the reconciler to iterate through the resources procedurally and offers a linear approach to solving the mapping problem. Furthermore, the process of undoing or deleting infrastructure is as simple as iterating through the set in reverse.

Rule 4: Make the Actual State Match the Expected State

The guarantee offered in the reconciler pattern is that the user gets exactly what was intended or an error. This is a guarantee that an engineer who is consuming the reconciler can rely on. This is important, as the consumer shouldn't have to concern themselves with validating that the reconciler mutation was idempotent and ended as expected. The implementation is ultimately responsible for addressing this concern. With the guarantee in place, using the reconciler pattern in more complex operations, such as a controller or operator, is now much simpler.

The implementation should, before returning to the calling code, check that the newly reconciled actual data structure matches the original expected data structure. If it does not, it should error. The consumer should never concern themselves with validating the API, and should be able to trust the reconciler to error if something goes wrong.

Because the data structures are immutable and the API will error if the reconciler pattern is not successful, we can put a high level of trust in the API. With complex

systems, it is important that you are able to trust that your software works or fails in predictable ways.

The Reconciler Pattern's Methods

With the information and rules of the reconciler patterns we just explained, let's look at how some of those rules have been implemented. We will do this by looking at the methods needed for an application that implements the reconciler pattern.

The first method of the reconciler pattern is `GetActual()`. This method is sometimes called an *audit* and is used to query for the actual state of infrastructure. The method works by generating a map of resources, then procedurally calling each resource to see what, if anything, exists. The method will update the data structure based on the queries and return a populated data structure that represents what is actually running.

A much simpler method, `GetExpected()`, will read the intended state of the world from the data store. In the case of the *infrastructure.yaml* example (Example 4-4), `GetExpected()` would simply unmarshal this YAML and return it in the form of the data structure in memory. No resource auditing is done at this step.

The most exciting method is the `Reconcile()` method, in which the reconciler implementation will be handed the actual state of the world, as well as the expected state of the world.

This is the core of the intent-driven behavior of the reconciler pattern. The underlying reconciler implementation would use the same resource mapping logic used in `GetActual()` to define a set of resources. The reconciler implementation would then operate on these resources, reconciling each one independently.

It is important to understand the complexity of each of these resource reconciliation steps. The reconciler implementation must work in two ways.

First, get the resource properties from the desired and actual state. Next, apply changes to the minimal set of properties to make the actual state match the desired state.

If at any time the two representations of infrastructure conflict, the reconciler implementation must take action and mutate the infrastructure. After the reconciliation step has been completed, the reconciler implementation must create a new representation and then move on to the next resource. After all the resources have been reconciled, the reconciler implementation returns a new data structure to the caller of the interface. This new data structure now accurately represents the actual state of the world and should have a guarantee that it matches the original actual data structure.

The final method of the reconciler pattern is the `Destroy()` method. The word `Destroy()` was intentionally chosen over `Delete()` because we want the engineer to be aware that the method should destroy infrastructure, and never disable it. The implementation of the `Destroy()` method is simple. It uses the same resource mapping as defined in the preceding implementation methods, but merely operates on the resources in reverse.

Example of the Pattern in Go

Example 4-7 is the reconciler pattern defined in four methods in the Go programming language.

Don't worry if you don't know Go. The pattern can easily be implemented in any language. We just use Go because it clearly defines the input and output type of each method. Please read the comments for each method, as it defines what each method needs to do, and when it should be used.

Example 4-7. The reconciler pattern interface

```
// The reconciler interface below is an example of the reconciler pattern.
// It should be used whenever a user intends on mutating infrastructure based on a
// state that might have changed over time.
type Reconciler interface {

    // GetActual takes no arguments for input and returns a populated data
    // structure as well as a possible error. The data structure should
    // contain a complete representation of the infrastructure.
    // This is sometimes called an audit. This method
    // should be used to get a real-time representation of what infrastructure is
    // in existence.
        GetActual() (*Api, error)

    // GetExpected takes no arguments for input and returns a populated data
    // structure that represents what infrastructure an operator has declared to
    // exist, as well as a possible error. This is sometimes called expected or
    // intended state. This method should be used to get a real-time representation
    // of what infrastructure an operator intends to be in existence.
        GetExpected() (*Api, error)

    // Reconcile takes two arguments.
    // actualApi is a populated data structure that is returned from the GetActual
    // method. expectedApi is a populated data structure that is returned from the
    // GetExpected method. Reconcile will return a populated data structure that is
    // a representation of the new "actual" state, as well as a possible error.
    // By definition, the data structure returned here should match
    // the data structure returned from the GetExpected method. This method is
    // responsible for making changes to infrastructure.
```

```
        Reconcile(actualApi, expectedApi *Api) (*Api, error)

    // Destroy takes one argument.
    // actualApi is a populated data structure that is returned from the GetActual
    // method. Destroy will return a populated data structure that is a
    // representation of the new "actual" state, as well as a possible error. By
    // definition, the data structure returned here should match
    // the data structure returned from the GetExpected method.
        Destroy(actualApi *Api) (*Api, error)
}
```

The Auditing Relationship

As time progresses, the last audit of our infrastructure becomes stale, increasing the risk that our representation of infrastructure is inaccurate. So the trade-off is that an operator can exchange frequency of audits for accuracy of infrastructure representation.

A reconciliation is implicitly an audit. If nothing has changed, the reconciler will detect that nothing needs to be done, and the operation becomes an audit, validating that our representation of our infrastructure is accurate.

Furthermore, if there happens to be something that has changed in our infrastructure, the reconciler will detect the change and attempt to correct it. Upon completion of the reconcile, the state of the infrastructure is guaranteed to be accurate. So implicitly, we have audited the infrastructure again.

Auditing and Reconciler Pattern in Configuration Management

Infrastructure engineers may be familiar with the reconciler pattern from configuration management tools, which use similar methods to mutate operating systems. The configuration management tool is passed a set of resources to manage from a set of manifests or recipes defined by the engineer.

The tool will then take action on the system to make sure the actual state and desired state match. If no changes are made, then a simple audit is performed to make sure the states match.

The reason configuration management is not the same thing as cloud native infrastructure applications is because configuration management traditionally abstracts single nodes and does not create or manage infrastructure resources.

Some configuration management tools are extending their use in this space to varying degrees of success, but they remain in the category of infrastructure as code and not the bidirectional relationship that infrastructure as software provides.

A lightweight and stable reconciler implementation can yield powerful results that are reconciled quickly, giving the operator confidence in accurate infrastructure representation.

Using the Reconciler Pattern in a Controller

Orchestration tooling such as Kubernetes offers a platform in which we can run applications conveniently. The idea of a controller is to serve a control loop for an intended state. Kubernetes is built on this fundamental. The reconciler pattern makes it easy to audit and reconcile objects controlled by Kubernetes.

Imagine a loop that would endlessly flow through the reconciler pattern in the following steps:

1. Call `GetExpected()` and read from a data store the intended state of infrastructure.
2. Call `GetActual()` and read from an environment to get the actual state of infrastructure.
3. Call `Reconcile()` and reconcile the states.

The program that implemented the reconciler pattern in this way would serve as a controller. The beauty of the pattern becomes immediately evident, since it's easy to see how small the program for the controller itself would have to be.

Furthermore, making a change to the infrastructure is as simple as mutating the state store. The controller will read the change the next time `GetExpected()` is called and trigger a reconcile. The operator in charge of the infrastructure can rest assured that a stable and reliable loop is running quietly in the background, enforcing her will across her infrastructure environment. Now an operator manages infrastructure by managing an application.

> The goal seeking behavior of the control loop is very stable. This has been proven in Kubernetes where we have had bugs that have gone unnoticed because the control loop is fundamentally stable and will correct itself over time.
>
> If you are edge triggered you run risk of compromising your state and never being able to re-create the state. If you are level triggered the pattern is very forgiving, and allows room for components not behaiving as they should to be rectified. This is what makes Kubernetes work so well.
>
> —Joe Beda, CTO of Heptio

Destroying infrastructure is now as simple as notifying the controller that we wish to destroy infrastructure. This could be done in a number of ways. One way would be to have the controller respect a disabled state file. This could be represented by flipping a bit from on to off.

Another way could be by deleting the content of the state. Regardless of how an operator chooses to signal a `Destroy()`, the controller is ready to call the convenient `Destroy()` method.

Conclusion

Infrastructure engineers are now software engineers, tasked with building advanced and highly distributed systems—and working backward. They must write software that manages the infrastructure they are responsible for.

While there are many similarities between the two disciplines, there is a lifetime of learning the trade of engineering infrastructure management applications. Hard problems, such as bootstrapping infrastructure, continually evolve and require engineers to keep learning new things. There is also an ongoing need to maintain and optimize infrastructure that is sure to keep engineers employed for a very long time.

The chapter has equipped the user with powerful patterns and fundamentals in mapping ambiguous API structures into granular resources. The resources can be applied in your local data center, on top of a private cloud, or in a public cloud.

Understanding the basics of how these patterns work is critical to building reliable infrastructure management applications. The patterns set out in this chapter are intended to give engineers a starting point and inspiration for building declarative infrastructure management applications.

There is no right or wrong answer in building infrastructure management applications, so long as the applications adhere to the Unix philosophy: "Do one thing. Do it well."

CHAPTER 5
Managing Cloud Native Applications

Cloud native applications are designed to be maintained by infrastructure. As we've shown in the previous chapters, cloud native infrastructure is designed to be maintained by applications.

With traditional infrastructure, the majority of the work to schedule, maintain, and upgrade applications is done by humans. This can include manually running services on individual hosts or defining a snapshot of infrastructure and applications in automation tools.

But if infrastructure can be managed by applications and at the same time manage the applications, then infrastructure tooling becomes just another application. The responsibilities engineers have with infrastructure can be expressed in the reconciler pattern and built into applications that run on that infrastructure.

We just spent the past three chapters explaining how we build applications that can manage infrastructure. This chapter will address how to run those applications, or any application, on the infrastructure.

As discussed earlier, it's important to keep your infrastructure and applications simple. A common way to manage complexity in applications is to break them apart into small, understandable components. We usually accomplish this by creating single-purpose services,[1] or breaking out code into a series of event-triggered functions.[2]

The proliferation of smaller, deployable units can be overwhelming for even the most automated infrastructure. The only way to manage a large number of applications is

1 Often called microservices or service-oriented architecture (SOA).

2 Called serverless or function as a service (FaaS).

to have them take on the operational functionally described in Chapter 1. The applications need to become cloud native before they can be managed at scale.

This chapter will not help you build the next great app, but it should give you some starting points in making your applications work well when running on cloud native infrastructure.

Application Design

There are many books that discuss how applications should be architected. This book is not intended to be one of them. However, it is still important to understand how application architecture influences effective infrastructure designs.

As we discussed in Chapter 1, we are going to assume the applications are designed to be cloud native because they gain the most benefits from cloud native infrastructure. Fundamentally, cloud native means the applications are designed to be managed by software, not humans.

The design of an application is a separate consideration from how it is packaged. Applications can be cloud native and packaged as an RPM or DEB files and deployed to VMs instead of containers. They can be monolithic or microservices, written in Java or Go.

These implementation details do not make an application designed to run in the cloud.

As an example, let's pretend we have an application written in Go and packaged in a container. We can even say the container runs on Kubernetes and would be considered a microservice by whatever definition you choose.

Is this pretend application "cloud native"?

What if the application logs all activity to a file and hardcodes the database IP address? Maybe it doesn't accept runtime configuration and stores state on the local disk. What if it doesn't exit in a predictable manner, or hangs and waits for a human to debug it?

This pretend application may appear cloud native from the chosen language and packaging, but it is most definitely not. A framework such as Kubernetes can help manage this application through various features, but even if you're able to make it run, the application is clearly designed to be maintained and run by humans.

Some of the features that make an application run better on cloud native infrastructure are explained in more detail in Chapter 1. If we have the features prescribed in Chapter 1, there is still another consideration for applications: how do we effectively manage them?

Implementing Cloud Native Patterns

Patterns such as resiliency, service discovery, configuration, logging, health checks, and metrics can all be implemented in applications in different ways. A common practice to implement these patterns is through standard language libraries imported into the applications. Netflix OSS and Twitter's Finagle are very good examples of implementing these features in Java language libraries.

When you use a library, an application can import the library, and it will automatically get many of these features without extra code. This model makes a lot of sense when there are few supported languages within an organization. It allows best practice to be the easy thing to do.

When organizations start implementing microservices, they tend to drift toward polyglot services.[3] This allows for freedom to choose the right language for the service, but makes it very difficult to maintain libraries for all the languages.

Another way to get some of these features is through what is known as the "sidecar" pattern. This pattern bundles processes with applications that implement the various management features. It is often implemented as a separate container, but you can also implement it by just running another daemon on a VM.

Examples of sidecars include the following:

Envoy proxy
: Adds resiliency and metrics to services

Registrator
: Registers services with an external service discovery

Configuration
: Watches for configuration changes, and notifies the service process to reload

Health endpoint
: Provide HTTP endpoints for checking the health of the application

Sidecar containers can even be used to adapt polyglot containers to expose language-specific endpoints to interact with applications that use libraries. Prana from Netflix does just that for applications that don't use their standard Java library.

Sidecar patterns make sense when centralized teams manage specific sidecar processes. If an engineer wants to expose metrics in their service, they can build it into the application—or a separate team can also provide a sidecar that processes logging output and exposes the calculated metrics for them.

3 That is, services written in many different languages.

In both cases, the service can have functionality added with less effort than rewriting the application. Once the ability to manage the application with software is available, let's look at how the application's life cycle should be managed.

Application Life Cycle

Life cycles for cloud native applications are no different than traditional applications, except their stages should be managed by software.

This chapter is not intended to explain all the patterns and options involved in managing applications. We will briefly discuss a few stages that particularly benefit from running cloud native applications on top of cloud native infrastructure: deploy, run, and retire.

These topics are not all inclusive of every option, but many other books and articles exist to explore the options, depending on the application's architecture, language, and chosen libraries.

Deploy

Deployments are one area where applications rely on infrastructure the most. While there is nothing stopping an application from deploying itself, there are still many other aspects that the infrastructure manages.

How you do integration and delivery are topics we will not address here, but a few practices in this space are clear. Application deployment is more than just taking code and running it.

Cloud native applications are designed to be managed by software in all stages. This includes ongoing health checks as well as initial deployments. Human bottlenecks should be eliminated as much as possible in the technology, processes, and policies.

Deployments for applications should be automated, self-service, and, if under active development, frequent. They should also be tested, verified, and uneventful.

Replacing every instance of an application at once is rarely the solution for new versions and features. New features are "gated" behind configuration flags, which can be selectively and dynamically enabled without an application restart. Version upgrades are partially rolled out, verified with tests, and, when all tests pass, rolled out in a controlled manner.

When new features are enabled or new versions deployed, there should exist mechanisms to control traffic toward or away from the application. This can limit outage impact and allows slow rollouts and faster feedback loops for application performance and feature usage.

The infrastructure should take care of all details of deploying software. An engineer can define the application version, infrastructure requirements, and dependencies, and the infrastructure will drive toward that state until it has satisfied all requirements or the requirements change.

Run

Running the application should be the most uneventful and stable stage of an application's life cycle. The two most important aspects of running software are discussed in Chapter 1: *observability* to understand what the application is doing, and *operability* to be able to change the application as needed.

We already went into detail in Chapter 1 about observability for applications by reporting health and telemetry data, but what do you do when things don't work as intended? If an application's telemetry data says it's not meeting the SLO, how can you troubleshoot and debug the application?

With cloud native applications, you should not SSH into a server and dig through logs. It may even be worth considering if you need SSH, log files, or servers at all.

You still need application access (API), log data (cloud logging), and servers somewhere in the stack, but it's worth going through the exercise to see if you need the traditional tools at all. When things break, you need a way to debug the application and infrastructure components.

When debugging a broken system, you should first look at your infrastructure tests. Testing should expose any infrastructure components that are not configured properly or are not providing the expected performance.

Just because you don't manage the underlying infrastructure doesn't mean the infrastructure cannot be the cause of your problems. Having tests to validate expectations will ensure your infrastructure is performing how you expect.

After infrastructure has been ruled out, you should look to the application for more information. The best places to turn for application debugging are application performance management (APM) and possibly distributed application tracing via standards such as OpenTracing.

OpenTracing examples, implementation, and APM are out of scope for this book. As a very brief overview, OpenTracing allows you to trace calls throughout the application to more easily identify network and application communication problems. An example visualization of OpenTracing can be seen in Figure 5-1. APM adds tools to your applications for reporting metrics and faults to a collection service.

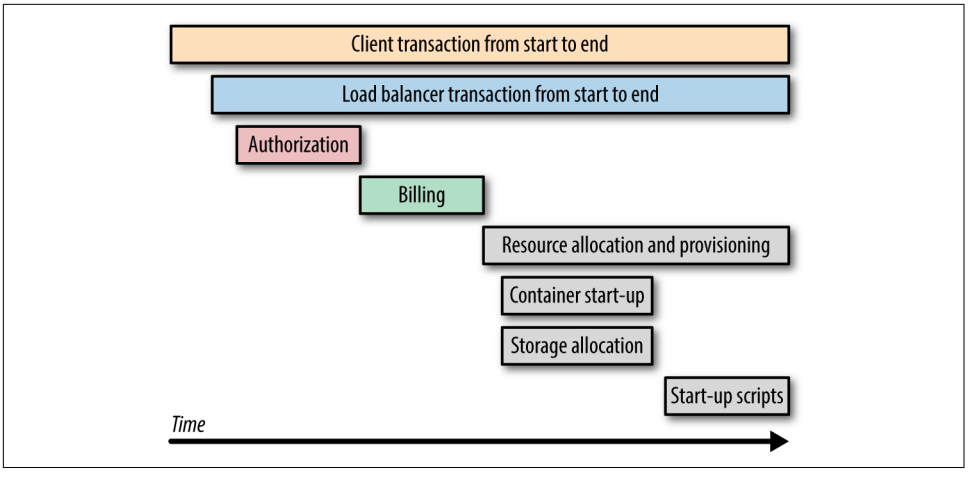


Figure 5-1. OpenTracing visualization

When tests and tracing still do not expose the problem, sometimes you just need to enable more verbose logging on the application. But how do you enable debugging without destroying the problem?

Configuration at runtime is important for applications, but in a cloud native environment, configuration should be dynamic without application restarts. Configuration options are still implemented via a library in the application, but flag values should have the ability to dynamically change through a centralized coordinator, application API calls, HTTP headers, or a myriad of ways.

Two examples of dynamic configuration are Netflix's Archaius and Facebook's Gatekeeper. Justin Mitchell, a former Facebook engineering manager, shared in a Quora post that:

> [Gatekeeper] decoupled feature releasing from code deployment. Features might be released over the course of days or weeks as we watched user metrics, performance, and made sure services were in place for it to scale.

Allowing dynamic control over application configuration enables more control over exposed features and better test coverage of deployed code. Just because pushing new code is easy doesn't mean it is the right solution for every situation.

Infrastructure can help solve this problem and enable more flexible applications by coordinating when features are enabled and routing traffic based on advanced network policies. This pattern also allows finer-grained controls and better coordination of roll-out or roll-back scenarios.

In a dynamic, self-service environment the number of applications that will get deployed will grow rapidly. You need to make sure you have an easy way to dynamically debug applications in a similar self-service model to deploy the applications.

As much as engineers love pushing new applications, it is conversely difficult to get them to retire old applications. Even still, it is a crucial stage in an application's life cycle.

Retire

Deploying new applications and services is common in a fast-moving environment. Retiring applications should be self-service in the same way as creating them.

If new services and resources are deployed and monitored automatically, they should also be retired under the same criteria. Deploying new services as quickly as possible with no removal of unused services is the easiest way to accrue technical debt.

Identifying services and resources that should be retired is business specific. You can use empirical data from your application's telemetry measurements to know if an application is being used, but the decision to retire applications should be made by the business.

Infrastructure components (e.g., VM instances and load balancer endpoints) should be automatically cleaned up when not needed. One example of automatic component cleanup is Netflix's Janitor Monkey. The company explains in a blog post:

> Janitor Monkey determines whether a resource should be a cleanup candidate by applying a set of rules on it. If any of the rules determines that the resource is a cleanup candidate, Janitor Monkey marks the resource and schedules a time to clean it up.

The goal in all of these application stages is to have infrastructure and software manage the aspects that would traditionally be managed by a human. Instead of writing automation scripts that run once by a human, we employ the reconciler pattern combined with component metadata to constantly run and make decisions about actions that need to be taken on a high level based on current context.

Application life cycle stages are not the only aspects where applications depend on infrastructure. There are also some fundamental services for which applications in every stage will depend on infrastructure. We will look at some of the supporting services and APIs infrastructure provides to applications in the next section.

Application Requirements on Infrastructure

Cloud native applications expect more from infrastructure than just executing a binary. They need abstractions, isolations, and guarantees about how they'll run and be managed. In return they are required to provide hooks and APIs to allow the infrastructure to manage them. To be successful, there needs to be a symbiotic relationship.

We defined cloud native applications in Chapter 1, and just discussed some life cycle requirements. Now let's look at more expectations they have from an infrastructure built to run them:

- Runtime and isolation
- Resource allocation and scheduling
- Environment isolation
- Service discovery
- State management
- Monitoring and logging
- Metrics aggregation
- Debugging and tracing

All of these should be default options for services or provided from self-service APIs. We will explain each requirement in more detail to make sure the expectations are clearly defined.

Application Runtime and Isolation

Traditional applications only needed a kernel and possibly an interpreter to run. Cloud native applications still need that, but they also need to be isolated from the operating system and other applications where they run. Isolation enables multiple applications to run on the same server and control their dependencies and resources.

Application isolation is sometimes called *multitenancy*. That term can be used for multiple applications running on the same server and for multiple users running applications in a shared cluster. The users can be running verified, trusted code, or they may be running code you have no control over and do not trust.

To be cloud native does not require using containers. Netflix pioneered many of the cloud native patterns, and when the company transitioned to running on a public cloud, it used VMs as their deployment artifact, not containers. FaaS services (e.g., AWS Lambda) are another popular cloud native technology for packaging and deploying code. In most cases, they use containers for application isolation but container packaging is hidden from the user.

What Is a Container?

There are many different implementations of containers. Docker popularized the term *container* to describe a way to package and run an application in an isolated environment. Fundamentally, containers use kernel primitives or hardware features to isolate processes on a single operating system.

Levels of container isolation can vary, but usually it means the application runs with an isolated root filesystem, namespaces, and resource allocation (e.g., CPU and RAM) from other processes on the same server. The container format has been adopted by many projects and has created the Open Container Initiative (OCI), which defines standards on how to package and run an application container.

Isolation also puts a burden on the engineers writing the application. They are now responsible for declaring all software dependencies. If they fail to do so, the application will not run because necessary libraries will not be available.

Containers are often chosen for cloud native applications because better tooling, processes, and orchestration tools have emerged for managing them. While containers are currently the easiest way to implement runtime and resource isolation, this has not (and likely will not) always be the case.

Resource Allocation and Scheduling

Historically, applications would provide rough estimates around minimum system requirements, and it was the responsibility of a human to figure out where the application could run.[4] Human scheduling can take a long time to prepare the operating system and dependencies for the application to run.

The deployment can be automated through configuration management and provisioning, but it still requires a human to verify resources and tag a server to run the application. Cloud native infrastructure relies on dependency isolation and allows applications to run wherever resources are available.

With isolation, as long as a system has available processing, storage, and access to dependencies, applications can be scheduled anywhere. Dynamic scheduling removes the human bottleneck from making decisions that are better left to machines. A cluster scheduler gathers resource information from all systems and figures out the best place to run the application.

4 Also known as "meat schedulers."

Having humans schedule application placement doesn't scale. Humans get sick, take vacations (or at least they should), and are generally bottlenecks. As scale and complexity increases, it also becomes impossible for a human to remember where applications are running.

Many companies try to scale by hiring more people. This exacerbates the problem because then scheduling needs to be coordinated between multiple people. Eventually, human scheduling resorts to keeping a spreadsheet (or similar solution) of where each application runs.

Dynamic scheduling doesn't mean operators have no control. There are still ways an operator can override or force a scheduling decision based on knowledge the scheduler may not have. Overrides and manual resource scheduling should be provided via an API and not a meeting request.

Solving these problems is one of the main reasons Google wrote its internal cluster scheduler named Borg. In the Borg research paper, Google points out that:

> Borg provides three main benefits: it (1) hides the details of resource management and failure handling so its users can focus on application development instead; (2) operates with very high reliability and availability, and supports applications that do the same; and (3) lets us run workloads across tens of thousands of machines effectively.

The role of a scheduler in any cloud native environment is much of the same. Fundamentally, it needs to abstract away the many machines and allow users to request resources, not servers.

Environment Isolation

When applications are made of many services, infrastructure needs to provide a way to have defined isolation with all dependencies. Separating dependencies is traditionally managed by duplicating servers, networks, or clusters into development or testing environments. Infrastructure should be able to logically separate dependencies through application environments without full cluster duplication.

Logically splitting environments allows for better utilization of hardware, less duplication of automation, and easier testing for the application. On some occasions, a separate testing environment is required (e.g., when low-level changes need to be made). However, application testing is not a situation where a full duplicate infrastructure should be required.

Environments can be traditional permanent dev, test, stage, and production, or they can be dynamic branch or commit-based. They can even be segments of the production environment with features enabled via dynamic configuration and selective routing to the instances.

Environments should consist of all the data, services, and network resources needed by the application. This includes things such as databases, file shares, and any external services. Cloud native infrastructure can create environments with very low overhead.

Infrastructure should be able to provision the environment however it's used. Applications should follow best practices to allow flexible configuration to support environments and discover the endpoints for supporting services through service discovery.

Service Discovery

Applications almost certainly depend on one or more services to provide business benefit. It is the responsibility of the infrastructure to provide a way for services to find each other on a per-environment basis.

Some service discovery requires applications to make an API call, while others do it transparently with DNS or network proxies. It does not matter what tool is used, but it's important that services use service discovery.

While service discovery is one of the oldest network services (i.e., ARP and DNS), it is often overlooked and not utilized. Statically defining service endpoints in a per-instance text file or in code is not scalable and not suitable for a cloud native environment. Endpoint registration should happen automatically when services are created and endpoints become available or go away.

Cloud native applications work together with infrastructure to discover their dependent services. These include, but are not limited to, DNS, cloud metadata services, or standalone service discovery tools (i.e., etcd and consul).

State Management

State management is how infrastructure can know what needs to be done, if anything, to an application instance. This is distinctly different from application life cycle because the life cycle applies to applications throughout their development process. States apply to instances as they are started and stopped.

It is the application's responsibility to provide an API or hook so it can check for its current state. The infrastructure's responsibility is to monitor the instance's current state and act accordingly.

The following are some application states:

- Submitted
- Scheduled
- Ready

- Healthy
- Unhealthy
- Terminating

A brief overview of the states and corresponding actions would be as follows:

1. An application is submitted to be run.
2. The infrastructure checks the requested resources and schedules the application.

   While the application is starting, it will provide a ready/not ready status.
3. The infrastructure will wait for a ready state and then allow for consumption of the applications resources (e.g., adding the instance to a load balancer).

   If the application is not ready before a specified timeout, the infrastructure will terminate it and schedule a new instance of the application.
4. Once the application is ready, the infrastructure will watch the liveness status and wait for an unhealthy status or until the application is set to no longer run.

There are more states than those listed. States need to be supported by the infrastructure if they are to be correctly checked and acted upon. Kubernetes implements application state management through events, probes, and hooks, but every orchestration platform should have similar application management capabilities.

A Kubernetes event is triggered when an application is submitted, scheduled, or scaled. Probes are used to check when an application is ready to serve traffic (readiness) and to make sure an application is healthy (liveness). Hooks are used for events that need to happen before or after processes start.

The state of an application instance is just as important as application life cycle management. Infrastructure plays a key role in making sure instances are available and acting on them accordingly.

Monitoring and Logging

Applications should never have to request to be monitored or logged; they are basic assumptions for running on the infrastructure. More importantly, configuration for monitoring and logging, if required, should be declarative as code in the same way that application resource requests are made. If you have all the automation to deploy applications but can't dynamically monitor services, there is still work to be done.

State management (i.e., process health checks) and logging deal with individual instances of an application. The logging system should be able to consolidate logs base on the applications, environments, tags, or any other useful metadata.

Applications should, as much as possible, not have single points of failure and should have multiple instances running. If an application has 100 instances running, the monitoring system should not trigger an alert if a single instance becomes unhealthy.

Monitoring looks holistically at applications and is used for debugging and verifying desired states.[5] Monitoring is different than alerting, because alerting should be triggered based on the metrics and SLO of the applications.

Metrics Aggregation

Metrics are required to know how applications behave when they're in a healthy state. They also can provide insight into what may be broken when they are unhealthy—and just like monitoring, metrics collecting should be requested as code as part of an application definition.

The infrastructure can automatically gather metrics around resource utilization,[6] but it is the application's responsibility to preset metrics for service-level indicators.

While monitoring and logging are application health checks, metrics provide the needed telemetry data. Without metrics, there is no way of knowing if the application is meeting service-level objectives to provide business value.

It may be tempting to pull telemetry and health check data from logs, but be careful, because logging requires post-processing and more overhead than application-specific metric endpoints.

When it comes to gathering metrics, you want as close to real-time data as possible. This requires a simple and low-overhead solution that can scale.

Logging should be used for debugging, and a delay for data processing should be expected.

Similarly to logging, metrics are usually gathered at an instance level and then composed together in aggregate to provide a view of a complete service instead of individual instances.

Once applications present a way to gather metrics, it is the infrastructure's job to scrape, consolidate, and store the metrics for analysis. Endpoints for gathering

5 Sometimes called "black-box" monitoring. This means you test the service as if you were an outside observer to a system you had no control over.

6 This data is already required to enforce quotas.

metrics should be configurable on a per-application basis, but the data formatting should be standardized so all metrics can be viewed in a single system.[7]

Debugging and Tracing

Applications are easy to debug during development. Integrated development environments (IDE), code break points, and running in debug mode are all tools the engineer has at his disposal when writing code.

Introspection is much more difficult for deployed applications. This problem is more acute when applications are composed of tens or hundreds of microservices or independently deployed functions. It may also be impossible to have tooling built into applications when services are written in multiple languages and by different teams.

The infrastructure needs to provide a way to debug a whole application and not just the individual services. Debugging can sometimes be done through the logging system, but reproducing bugs requires a shorter feedback loop.

Debugging is a good use of dynamic configuration, discussed earlier. When issues are found, applications can be switched to verbose logging, without restarting, and traffic can be routed to the instances selectively through application proxies.

If the issue cannot be resolved through log output, then distributed tracing provides a different interface to visualize what is happening. Distributed tracing systems such as OpenTracing can complement logs to help humans debug problems.

Tracing provides shorter feedback loops for debugging distributed systems. If it cannot be built into applications, it can be done transparently by the infrastructure through proxies or traffic analysis. When you are running any coordinated applications at scale, it is a requirement that the infrastructure provides a way to debug applications.

While there are many benefits and implementation details for setting up tracing in a distributed system, we will not discuss them here. Application tracing has always been important, and is increasingly difficult in a distributed system. Cloud native infrastructure needs to provide tracing that can span multiple services in a transparent way.

7 It's helpful to have a single metrics gathering system to find correlation between seemingly disjoint services that may impact each other in unexpected ways.

Conclusion

The applications requirements have changed: a server with an operating system and package manager is no longer enough. Applications now require coordination of services and higher levels of abstraction. The abstractions allow resources to be separated from servers and consumed programmatically as needed.

The requirements laid out in this chapter are not all the services that infrastructure can provide, but they are the basis for what cloud native applications expect. If the infrastructure does not provide these services, then applications will have to implement them, or they will fail to reach the scale and velocity required by modern business.

Infrastructure won't evolve on its own; people need to change their behavior and fundamentally think of what it takes to run an application a different way. Luckily there are projects that build on experience from companies that have pioneered these solutions.

Applications depend on the features and services of infrastructure to support agile development. Infrastructure requires applications to expose endpoints and integrations to be managed autonomously. Engineers should use existing tools when possible and build with the goal of designing resilient, simple solutions.

About the Authors

Justin Garrison is an engineer at one of the world's largest media companies. He loves open source almost as much as he loves community. He is not a fan of buzzwords but searches for the patterns and benefits behind technology trends. He frequently shares his findings and tries to disseminate knowledge through practical lessons and unique examples. He is an active member in many communities and constantly questions the status quo. He is relentless in trying to learn new things and giving back to the communities who have taught him so much.

Kris Nova is a senior developer advocate for Microsoft with an emphasis in containers and the Linux operating system. She lives and breathes open source. She believes in advocating for the best interest of the software and keeping the design process open and honest. She is a backend infrastructure engineer, with roots in Linux and C. She has a deep technical background in the Go programming language and has authored many successful tools in Go. She is a Kubernetes maintainer and the creator of kubicorn, a successful Kubernetes infrastructure management tool. She organizes a special interest group in Kubernetes and is a leader in the community. Kris understands the grievances with running cloud native infrastructure via a distributed cloud native application.

Colophon

The animal on the cover of *Cloud Native Infrastructure* is an Andean condor (*Vultur gryphus*). As the name implies, this New World vulture inhabits South America's Pacific coast, extending into the Andes. Weighing up to 33 pounds, it's the largest bird capable of flight, with a 10-foot wingspan that helps it glide on ocean breezes and mountainous thermal currents. This carnivorous bird is a scavenger, and prefers the carcasses of large animals, such as horses, cattle, llamas, and sheep.

The Andean condor has a hulking, menacing appearance. Its plumage is black except for a regal white ruffle around the neck. Like other vultures, this bird has a bald (featherless) head, which is dark red. The male is distinguished by a large red comb. During the male's courtship dance, its neck inflates and changes to bright yellow to attract the female's attention.

The Andean condor mates for life and can live for 50 years or more in the wild (up to 75 in captivity). It nests at high elevations, and produces only one or two eggs every other year; the young are raised by both parents until age two.

The Andean condor is used as a national symbol in South American countries such as Peru, Argentina, and Chile, similar to the bald eagle in the United States.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *Museum of Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.