

O'REILLY®



Cloud Native Infrastructure

PATTERNS FOR SCALABLE INFRASTRUCTURE AND APPLICATIONS
IN A DYNAMIC ENVIRONMENT

Justin Garrison & Kris Nova

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 Heptio with an emphasis in containers, infrastructure, and Kubernetes. She is an ambassador for the Cloud Native Computing Foundation.

“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

US \$39.99

CAN \$52.99

ISBN: 978-1-491-98430-7



Twitter: @oreillymedia
facebook.com/oreilly

Cloud Native Infrastructure

*Patterns for Scalable Infrastructure and
Applications in a Dynamic Environment*

Justin Garrison and Kris Nova

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Cloud Native Infrastructure

by Justin Garrison and Kris Nova

Copyright © 2018 Justin Garrison and Kris Nova. All rights reserved.

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

2018-02-14: Second Release

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

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Cloud Native Infrastructure*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-98430-7

[LSI]

Table of Contents

Introduction.....	vii
1. What Is Cloud Native Infrastructure?.....	1
Cloud Native Benefits	2
Servers	3
Virtualization	3
Infrastructure as a Service	4
Platform as a Service	4
Cloud Native Infrastructure	6
What Is Not Cloud Native Infrastructure?	7
Cloud Native Applications	9
Microservices	10
Health Reporting	10
Telemetry Data	12
Resiliency	13
Declarative, Not Reactive	16
How Do Cloud Native Applications Impact Infrastructure?	17
Conclusion	17
2. When to Adopt Cloud Native.....	19
Applications	20
People	21
Systems	21
Business	23
When You Don't Need Cloud Native Infrastructure	24
Technical Limitations	24
Business Limitations	26
Conclusion	27

3. Evolution of Cloud Native Deployments.....	29
Representing Infrastructure	30
Infrastructure as a Diagram	30
Infrastructure as a Script	32
Infrastructure as Code	34
Infrastructure as Software	36
Deployment Tools	38
Idempotency	40
Handling Failure	40
Conclusion	42
4. Designing Infrastructure Applications.....	43
The Bootstrapping Problem	44
The API	45
The State of the World	45
The Reconciler Pattern	49
Rule 1: Use a Data Structure for All Inputs and Outputs	50
Rule 2: Ensure That the Data Structure Is Immutable	50
Rule 3: Keep the Resource Map Simple	52
Rule 4: Make the Actual State Match the Expected State	53
The Reconciler Pattern’s Methods	54
Example of the Pattern in Go	55
The Auditing Relationship	56
Using the Reconciler Pattern in a Controller	57
Conclusion	58
5. Developing Infrastructure Applications.....	59
Designing an API	59
Adding Features	60
Deprecating Features	61
Mutating Infrastructure	63
Conclusion	66
6. Testing Cloud Native Infrastructure.....	67
What Are We Testing?	68
Writing Testable Code	69
Validation	69
Entering Your Codebase	70
Self-Awareness	72
Types of Tests	73
Infrastructure Assertions	73
Integration Testing	76

Unit Testing	76
Mock Testing	77
Chaos Testing	78
Monitoring Infrastructure	84
Conclusion	84
7. Managing Cloud Native Applications.....	87
Application Design	88
Implementing Cloud Native Patterns	89
Application Life Cycle	90
Deploy	90
Run	91
Retire	93
Application Requirements on Infrastructure	93
Application Runtime and Isolation	94
Resource Allocation and Scheduling	95
Environment Isolation	96
Service Discovery	97
State Management	97
Monitoring and Logging	98
Metrics Aggregation	99
Debugging and Tracing	100
Conclusion	101
8. Securing Applications.....	103
Policy as Code	103
Deployment Gating	104
Conformity Testing	106
Compliance Testing	107
Activity Testing	108
Auditing Infrastructure	109
Immutable Infrastructure	111
Conclusion	112
9. Implementing Cloud Native Infrastructure.....	113
Where to Focus for Change	114
People	114
Architecture	115
Chaos Management	117
Applications	118
Predicting the Future	119
Conclusion	120

A. Patterns for Network Resiliency	123
B. Lock-in	129
C. Box: Case Study	133
Index	137

Introduction

Technology infrastructure is at a fascinating point in its history. Due to requirements for operating at tremendous scale, it has gone through rapid disruptive change. The pace of innovation in infrastructure has been unrivaled except for the early days of computing and the internet. These innovations make infrastructure faster, more reliable, and more valuable.

The people and companies who have pushed the boundaries of infrastructure to its limits have found ways of automating and abstracting it to extract more business value. By offering a flexible, consumable resource, they have turned what was once an expensive cost center into a required business utility.

However, it is rare for utilities to provide financial value to the business, which means infrastructure is often ignored and seen as an unwanted cost. This leaves it with little time and money to invest in innovations or improvements.

How can such an essential and fascinating part of the business stack be so easily ignored? The business obviously pays attention when infrastructure breaks, so why is it so hard to improve?

Infrastructure has reached a maturity level that has made it boring to consumers. However, its potential and new challenges have ignited a passion in implementors and engineers.

Scaling infrastructure and enabling new ways of doing business have aligned engineers from all different industries to find solutions. The power of open source software (OSS) and communities driven to help each other have caused an explosion of new concepts and innovations.

If managed correctly, challenges with infrastructure and applications today will not be the same tomorrow. This allows infrastructure builders and maintainers to make progress and take on new, meaningful work.

Some companies have surmounted challenges such as scalability, reliability, and flexibility. They have created projects that encapsulate patterns others can follow. The patterns are sometimes easily discovered by the implementor, but in other cases they are less obvious.

In this book we will share lessons from companies at the forefront of cloud native technologies to allow you to conquer the problem of reliably running scalable applications. Modern business moves very fast. The patterns in this book will enable your infrastructure to keep up with the speed and agility demands of your business. More importantly, we will empower you to make your own decisions about when you need to employ these patterns.

Many of these patterns have been exemplified in open source projects. Some of those projects are maintained by the Cloud Native Computing Foundation (CNCF). The projects and foundation are not the sole embodiment of the patterns, but it would be remiss of you to ignore them. Look to them as examples, but do your own due diligence to vet every solution you employ.

We will show you the benefits of cloud native infrastructure and the fundamental patterns that make scalable systems and applications. We'll show you how to test your infrastructure and how to create flexible infrastructure that can adapt to your needs. You'll learn what is important and how to know what's coming.

May this book inspire you to keep moving forward to more exciting opportunities, and to share freely what you have learned with your communities.

Who Should Read This Book

If you're an engineer developing infrastructure or infrastructure management tools, this book is for you. It will help you understand the patterns, processes, and practices to create infrastructure intended to be run in a cloud environment. By learning how things should be, you can better understand the application's role and when you should build infrastructure or consume cloud services.

Application engineers can also discover which services should be a part of their applications and which should be provided from the infrastructure. Through this book they will also discover the responsibilities they share with the engineers writing applications to manage the infrastructure.

Systems administrators who are looking to level up their skills and take a more prominent role in designing infrastructure and maintaining infrastructure in a cloud native way can also learn from this book.

Do you run all of your infrastructure in a public cloud? This book will help you know when to consume cloud services and when to build your own abstractions or services.

Run a data center or on-premises cloud? We will outline what modern applications expect from infrastructure and will help you understand the necessary services to utilize your current investments.

This book is not a how-to and, outside of giving implementation examples, we're not prescribing a specific product. It is probably too technical for managers, directors, and executives but could be helpful, depending on the involvement and technical expertise of the person in that role.

Most of all, please read this book if you want to learn how infrastructure impacts business, and how you can create infrastructure proven to work for businesses operating at a global internet scale. Even if you don't have applications that require scaling to that size, you will still be better able to provide value if your infrastructure is built with the patterns described here, with flexibility and operability in mind.

Why We Wrote This Book

We want to help you by focusing on patterns and practices rather than specific products and vendors. Too many solutions exist without an understanding of what problems they address.

We believe in the benefits of managing cloud native infrastructure via cloud native applications, and we want to prescribe the ideology to anyone getting started.

We want to give back to the community and drive the industry forward. The best way we've found to do that is to explain the relationship between business and infrastructure, shed light on the problems, and explain the solutions implemented by the engineers and organizations who discovered them.

Explaining patterns in a product-agnostic way is not always easy, but it's important to understand why the products exist. We frequently use products as examples of patterns, but only when they will aid you in providing implementation examples of the solutions.

We would not be here without the countless hours people have volunteered to write code, help others, and invest in communities. We love and are thankful for the people that have helped us in our journey to understand these patterns, and we hope to give back and help the next generation of engineers. This book is our way of saying thank you.

Navigating This Book

This book is organized as follows:

- **Chapter 1** explains what cloud native infrastructure is and how we got where we are.
- **Chapter 2** can help you decide if and when you should adopt the patterns prescribed in later chapters.
- Chapters **3** and **4** show how infrastructure should be deployed and how to write applications to manage it.
- **Chapter 5** teaches you how to design reliable infrastructure from the start with testing.
- Chapters **6** and **7** show what managing infrastructure and applications looks like.
- **Chapter 8** wraps up and gives some insight into what's ahead.

If you're like us, you don't read books from front to back. Here are a few suggestions on broader book themes:

- If you are an engineer focused on creating and maintaining infrastructure, you should probably read Chapters **3** through **6** at a minimum.
- Application developers can focus on Chapters **4**, **5**, and **7**, about developing infrastructure tooling as cloud native applications.
- Anyone not building cloud native infrastructure will most benefit from Chapters **1**, **2**, and **8**.

Online Resources

You should familiarize yourself with the Cloud Native Computing Foundation (CNCF) and projects it hosts by visiting the **CNCF website**. Many of those projects are used throughout the book as examples.

You can also get a good overview of where the projects fit into the bigger picture by looking at the **CNCF landscape project** (see **Figure P-1**).

Cloud native applications got their start with the definition of Heroku's 12 factors. We explain how they are similar, but you should be familiar with what the 12 factors are (see <http://12factor.net>).

There are also many books, articles, and talks about DevOps. While we do not focus on DevOps practices in this book, it will be difficult to implement cloud native infrastructure without already having the tools, practices, and culture DevOps prescribes.

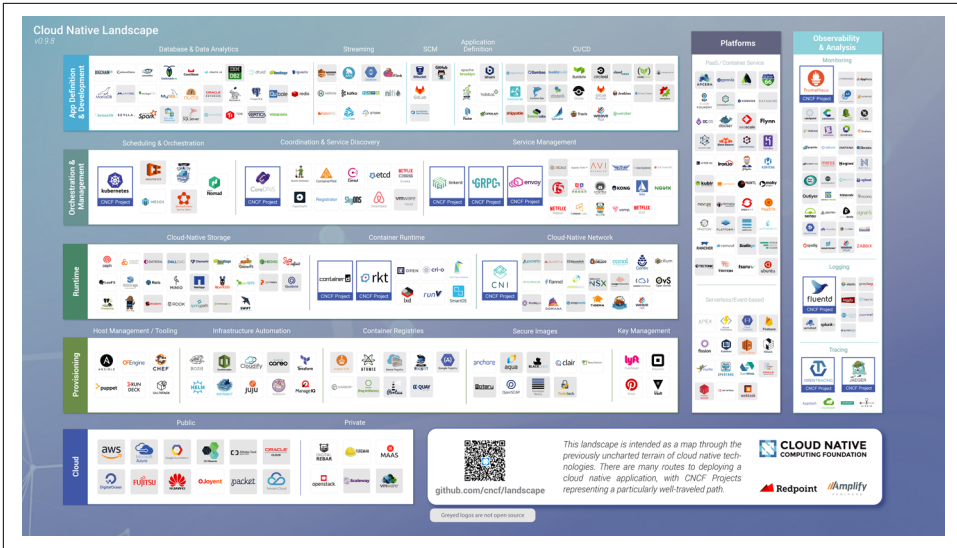


Figure P-1. CNCF landscape

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.




This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

O'Reilly Safari

 **Safari**[®] *Safari* (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit <http://oreilly.com/safari>.

How to Contact Us

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

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at http://bit.ly/cloud_native_infrastructure_1e.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

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

Acknowledgments

Justin Garrison

Thank you to Beth, Logan, my friends, family, and coworkers who supported us during this process. Thank you to the communities and community leaders who taught us so much and to the reviewers who gave valuable feedback. Thanks to Kris for making this book better in so many ways, and to you, the reader, for taking time to read books and improve your skills.

Kris Nova

Thanks to Allison, Bryan, Charlie, Justin, Kjersti, Meghann, and Patrick for putting up with my crap long enough for me to write this book. I love you, and am forever grateful for all you do.

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. See [Appendix B](#) for more information about different types of lock-in and what you should do about it.

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. See [Appendix C](#) for one example.

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 hypervisor² 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 tools³ 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.

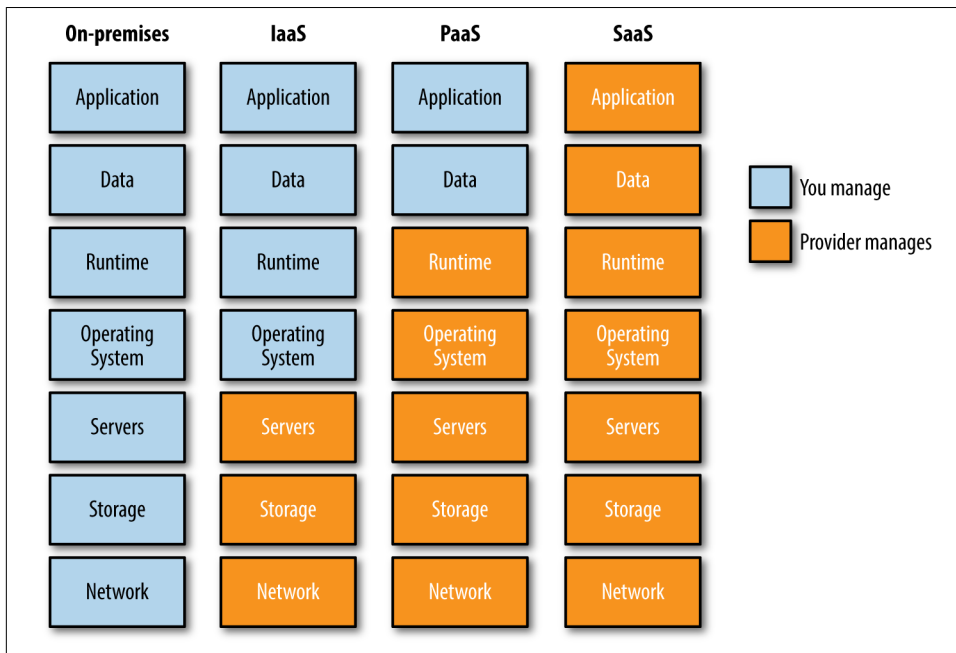


Figure 1-1. Infrastructure layers

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 cloud⁴ 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-parsable 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 run⁵ 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, which we address in [Appendix A](#).

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 health checks include Zookeeper's `ruok` command⁷ 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 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. For more information on building a resilient network layer, please see [Appendix A](#).

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 (see [Appendix A](#)), 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.

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 an 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.

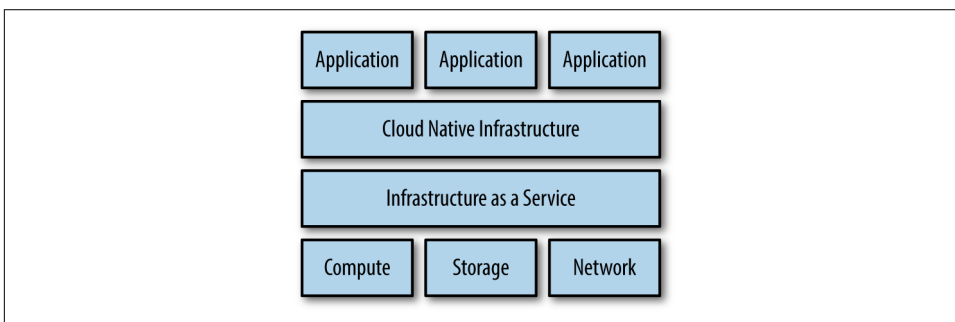


Figure 2-1. Example layers for cloud native infrastructure

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 [Appendix C](#) of 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. We will discuss when to consume hosted services in more detail in [Appendix B](#).

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.

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.

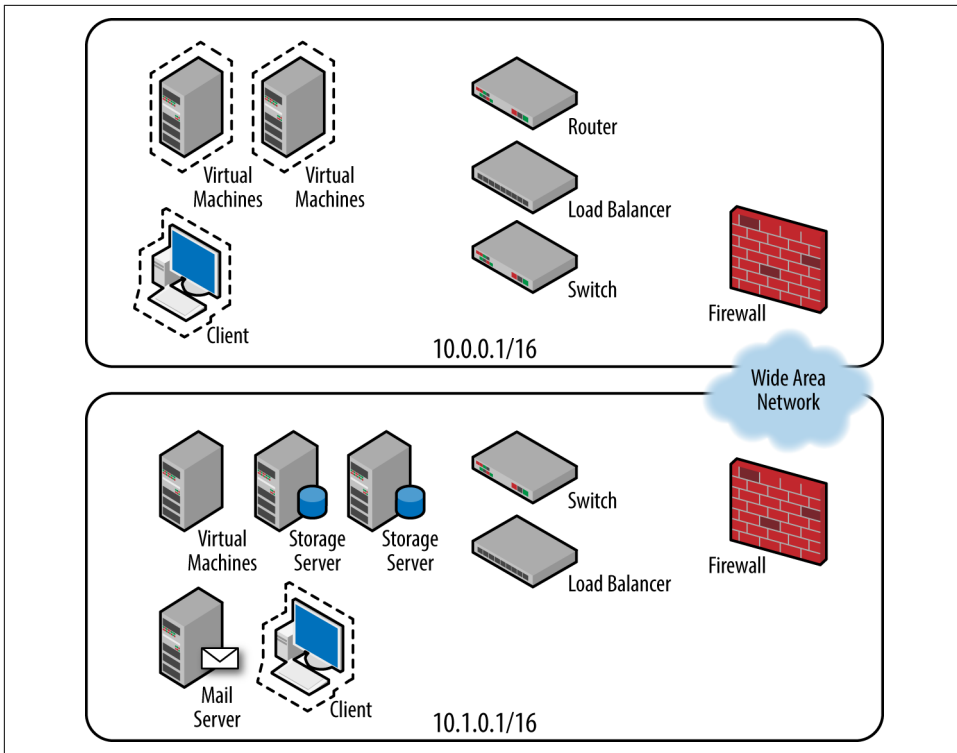


Figure 3-1. Simple infrastructure diagram

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.³ Having a human-readable and machine-parsible representation of infrastructure architecture, not just independent resource declaration, changed the industry forever.

³ For more information about lock-in, see [Appendix B](#).

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 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.⁴

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.

⁴ 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.⁵

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.

⁵ See [Appendix A](#) for examples on how to make network connections more resistant to failures.

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.

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 need 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.² 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.

² See [Chapter 8](#) on testing for even more reasons this is important.

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](#).

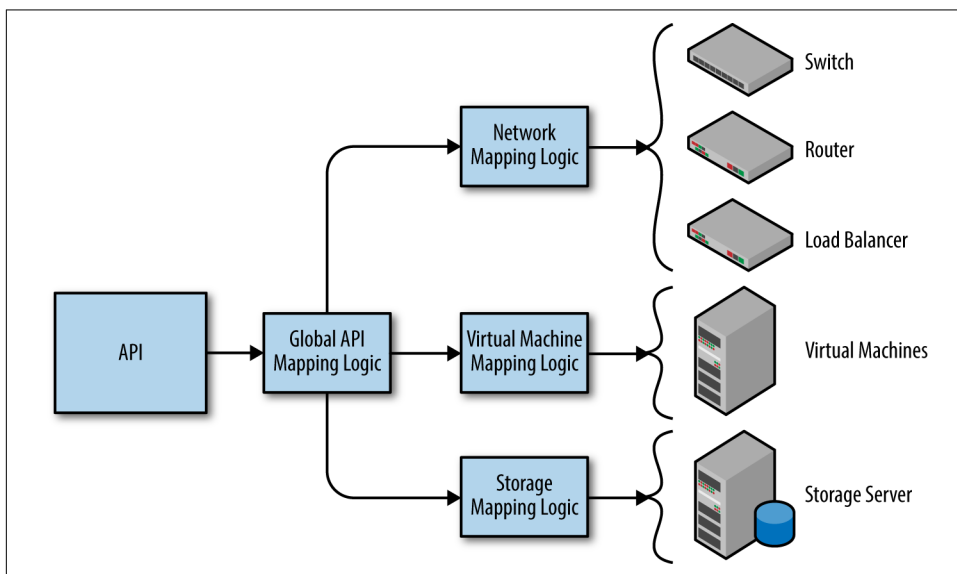


Figure 4-1. Diagram to map a structure to resources

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 behaving 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.”

Developing Infrastructure Applications

When building applications to manage infrastructure, you need to consider what APIs you will expose as much as what applications you will create. The APIs will represent the abstractions for your infrastructure, while the applications provide and consume APIs in the infrastructure.

It is important to have a firm grasp on why both are important and how you can use them to your advantage in creating scalable, resilient infrastructure.

In this chapter we will give a fictional example of a cloud native application and API that go through normal cycles for an application. If you want more information on managing cloud native applications, please see [Chapter 7](#).

Designing an API



The term *API* here is dealing with the infrastructure representation in a data structure and not concerned with how that representation is exposed or consumed. It is common to use an HTTP RESTful endpoint to communicate data structures, but the implementation is not important for this chapter.

Evolving infrastructure requires evolving the applications that support the infrastructure. The feature set for these applications will change over time, and thus infrastructure will implicitly evolve. As infrastructure continues to evolve, so must the applications that manage it.

Features, needs, and new advances in infrastructure will never stop. If we're lucky, the cloud provider APIs will be stable and not change frequently. As infrastructure

engineers, we need to be prepared to react appropriately to these needs. We need to be ready to evolve our infrastructure and the applications that support it.

We must create applications that can be scaled and also be ready to scale them. In order to do this, we need to understand the nuances of making large changes to our applications without breaking the existing flow of the application.

The beauty of engineering applications that manage infrastructure is that it liberates the operator from the opinions of others.

The abstractions used in an application are now up to the engineer to craft. If an API needs to be more literal, it can be; or if it needs to be opinionated and heavily abstracted, it can be. Powerful combinations of literal and abstracted definitions can give operators exactly what they want and need for managing infrastructure.

Adding Features

Adding a feature to an infrastructure application could be very simple or quite complex, depending on the nature of the feature. The goal of adding a feature is that we should be able to add new functionality without jeopardizing existing functionality. We never want to introduce a feature that will impact other components of the system in a negative way. Furthermore, we always want to make sure input into the system remains valid for a reasonable amount of time.

Example 5-1 is a concrete example of evolving an infrastructure API described earlier in the book. Let's call this first version of the API v1.

Example 5-1. v1.json

```
{
  "virtualMachines": [{
    "name": "my-vm",
    "size": "large",
    "localIp": "10.0.0.111",
    "subnet": "my-subnet"
  }],
  "subnets": [{
    "name": "my-subnet",
    "cidr": "10.0.100.0/24"
  }]
}
```

Imagine that we want to implement a feature that allows infrastructure operators to define DNS records for virtual machines. The new API would look slightly different. In **Example 5-2**, we will define a top-level directive called `version`, which will let our application know that this is v2 of the API. We will also add a new block that defines

the DNS record from within the context of the virtual machine block. This is a new directive that was not supported in v1.

Example 5-2. v2.json

```
{
  "version": "2",
  "virtualMachines": [{
    "name": "my-vm",
    "size": "large",
    "localIp": "10.0.0.111",
    "subnet": "my-subnet",
    "dnsRecords": [{
      "type": "A",
      "ttl": 60,
      "value": "my-vm.example.com"
    }]
  }],
  "subnets": [{
    "name": "my-subnet",
    "cidr": "10.0.100.0/24"
  }]
}
```

Both of these objects are valid, and the application should continue to support both of them. The application should detect that the v2 object intends to use the new DNS feature that was built into the application. The application should be smart enough to navigate the new feature gracefully. When resources are applied to the cloud, the new v2 object's resource set will be identical to the first v1 object but with the addition of a single DNS resource.

This begs an interesting question: what should the application do with older API objects? The application should still create the resources in the cloud, but support DNS-less virtual machines.

Over time an operator can modify existing virtual machine objects to use the new DNS feature. The application will naturally detect a delta and create a DNS record for the new feature.

Deprecating Features

Let's fast-forward to the next API version, v3. In this scenario, our API has evolved and we have reached a stalemate with how we are representing IP addresses.

In the first version of the API, v1, we were able to conveniently declare a local IP address for a network interface via the `localIp` directive. We have been tasked with

supporting multiple network interfaces for our virtual machines. It is important to note that this will conflict with the initial v1 API.

Let's take a look at the new API for v3 in [Example 5-3](#).

Example 5-3. v3.json

```
{
  "version": "2",
  "virtualMachines": [{
    "name": "my-vm",
    "size": "large",
    "networkInterfaces": [{
      "type": "local",
      "ip": "10.0.0.11"
    }],
    "subnet": "my-subnet",
    "dnsRecords": [{
      "type": "A",
      "ttl": 60,
      "value": "my-vm.example.com"
    }]
  }],
  "subnets": [{
    "name": "my-subnet",
    "cidr": "10.0.100.0/24"
  }]
}
```

With the new data structure that is necessary to define multiple network interfaces, we have deprecated the `localIp` directive. But we haven't removed the concept of defining an IP address, we have simply restructured it. This means we can begin to deprecate the directive in two stages. First we *warn*, and then we *deny*.

In the *warning* stage our application can output a fairly nasty warning about the `localIp` directive no longer being supported. The application can accept the directive as defined in the object, and translate the old API version v2 to the new API version v3 for the user.

The translation will take the value defined for `localIp` and create a single block within the new `networkInterface` directive that matches the initial value. The application can move forward with processing the API object as if the user had sent a v3 object instead of a v2 object. The user would be expected to heed the warning that the directive is being deprecated and update their representation in a timely manner.

In the *denial* stage our application will outright reject the v2 API. The user would be forced to update their API to the newer version, or risk casualties in their infrastructure.



Deprecating is dangerous

This is an extremely risky process, and navigating it successfully can be quite challenging. Rejecting input should only be done for a good reason.

If an input will break a guarantee in your application, it should be rejected. Otherwise, it is usually best practice to warn, and move on.

Breaking a user's input is an easy way to upset and possibly demoralize your operators.

The infrastructure engineer who is versioning the API will have to use their best judgment on when it is appropriate to deprecate features. In addition, the engineer will need to spend time trying to come up with clever solutions in which warning or translating would be appropriate. In some cases a silent translation is a monumental win in evolving cloud native infrastructure.

Mutating Infrastructure

Infrastructure will need to change or mutate over time. This is the nature of cloud native environments. Not only do applications have frequent deployments but the cloud providers where the infrastructure runs are always changing.

Infrastructure change can come in many forms, such as scaling infrastructure up or down, duplicating entire environments, or consuming new resources.

When an operator is tasked with mutating infrastructure, the true value of the API can be observed. Let's say we wanted to scale up the number of virtual machines in our environment. There would be no API version change needed, but a few minor tweaks to the representation and our infrastructure will soon reflect our change. It's that simple.

However, it is important to remember that the operator in this case could be a human, or very well could be *another* piece of software.

Remember, we intentionally structured our API to be easily decoded by computers. We can use software on both sides of the API!

Consuming and Producing APIs with Operators

CoreOS, Inc., a company building cloud native products and platforms, coined the term *operators*, which are Kubernetes controllers that replace the need for human involvement for managing specific applications. They do this by reconciling the intended state, as well as setting the intended state.

CoreOS described operators in their blog post this way:

An Operator is an application-specific controller that extends the Kubernetes API to create, configure, and manage instances of complex stateful applications on behalf of a Kubernetes user. It builds upon the basic Kubernetes resource and controller concepts but includes domain or application-specific knowledge to automate common tasks.

The pattern dictates that the operator can make changes to an environment by being given declarative instruction sets. An operator is a perfect example of the type of cloud native applications engineers should be creating to manage their infrastructure.

An easy scenario to envision is that of an autoscaler. Let's say we have a very simple piece of software that checks load average on virtual machines in an environment. We could define a rule that says whenever the mean load average gets above .7 we need to create more virtual machines to spread our load evenly.

The operator's rule would be tripped as the load average increased, and ultimately the operator would need to update the infrastructure API with another virtual machine. This manages scaling up our infrastructure, but just as easily we could define another rule to scale down when load average drops below .2. Note that the term *operator* here should be an application, not a person.

This is a very primitive example of autoscaling, but the pattern clearly demonstrates that software can now begin to play the role of a human operator.



There are many tools that can help with spreading application load over infrastructure such as Kubernetes, Nomad, and Mesos. This assumes the application layer is running an orchestrator that will manage this for us.

To take the value of the infrastructure API one step further, imagine if more than one infrastructure management application consumed the same API. This is an extremely powerful evolutionary paradigm in infrastructure.

Let's take the same API—which, remember, is just a few kilobytes of data—and run it against two independent infrastructure management applications. [Figure 5-1](#) shows an example of how two infrastructure applications can get data from the same API but deploy infrastructure to two separate cloud environments.

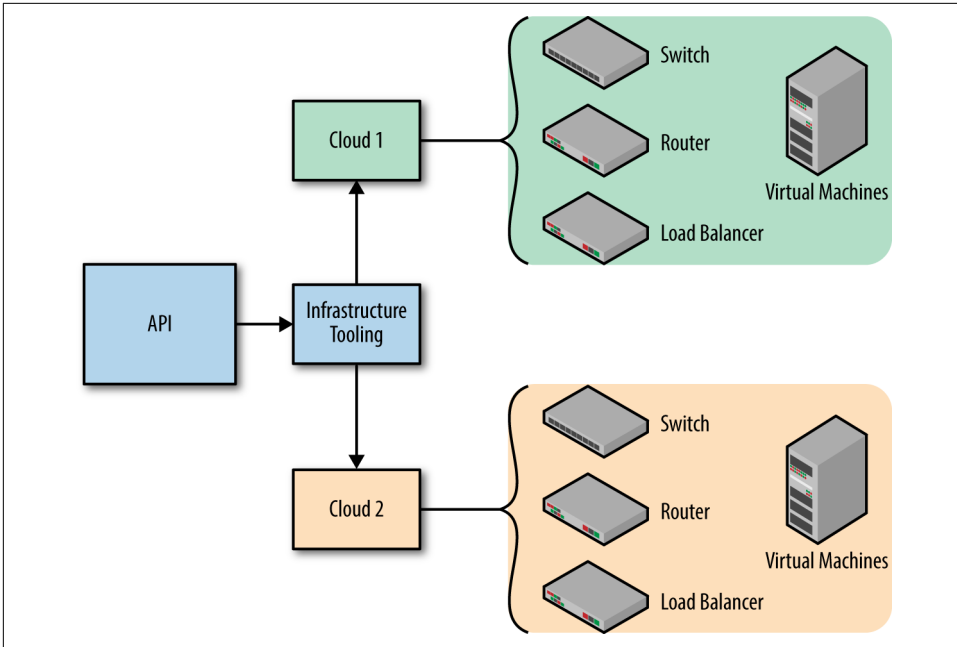


Figure 5-1. A single API being deployed in two clouds

This model gives infrastructure engineers fantastic power in being able to provide a common abstraction for multiple cloud providers. We can see how the application that is ensuring the API is now representing infrastructure in multiple places. The infrastructure does not have to be tied to the abstractions of a single cloud provider if the infrastructure API is responsible for providing its own abstractions and provisioning the resources. Users can create unique permutations of infrastructure in a cloud of their choosing.



Maintaining Cloud Provider Compatibility

While maintaining the API compatibility with the cloud provider will be a lot of work, it is minimal when it comes to changing your deployment workflows and provisioning processes. Remember, humans are harder to change than technology. If you can maintain a consistent environment for humans, it will offset the technical overhead required.

You should also weigh the benefit of multi-cloud compatibility. If it is not a requirement for your infrastructure, you can save a lot of engineering effort. See [Appendix B](#) when considering lock-in.

We can also speculate about having different infrastructure management applications running in the same cloud. Each of these applications might interpret the API differently and result in slightly different infrastructure. Depending on the intent of the operator defining the infrastructure, switching between management applications might be just the thing we need. [Figure 5-2](#) shows two applications reading the same API source, but implementing the data differently depending on environment and need.

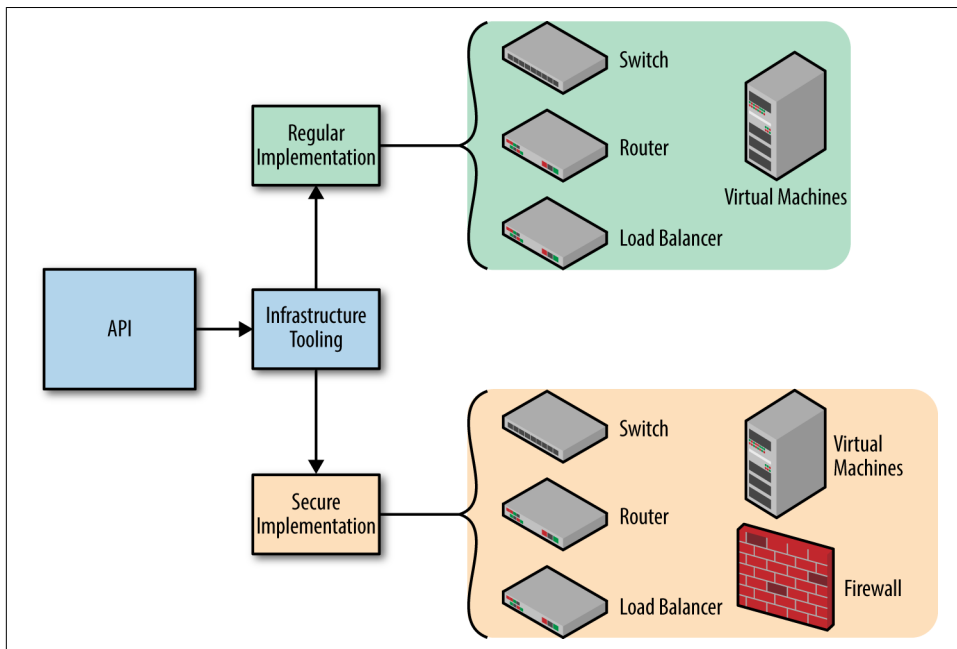


Figure 5-2. A single API being deployed in the same cloud in different ways

Conclusion

The permutations of infrastructure applications compared to infrastructure APIs are endless. This offers an extremely flexible and scalable solution for infrastructure engineers hoping to master their infrastructure in different environments and in different ways.

The various applications we might build in order to satisfy infrastructure requirements now become the representation of infrastructure itself. This is the epitome of infrastructure as software as defined in [Chapter 3](#).

It is important to remember that the applications we have been building are within themselves cloud native applications. This is an interesting twist in the story, as we are building cloud native applications to manage cloud native infrastructure.

Testing Cloud Native Infrastructure

Infrastructure supports applications. Being able to trust software is critical to success in engineering. What if every time you typed the `ls` command in a terminal a random action happened? You would never trust `ls` and would find a different way to list files in a directory.

We need to be able to trust our infrastructure. This chapter is aimed at opening up the ideologies on trusting and verifying infrastructure. The practices we will describe are designed to increase confidence in infrastructure for applications and engineers.

The practice of testing software is quite common in the software engineering space today. The practice of testing infrastructure, however, is virtually undefined.

This means that of all lessons in this book, this should be the most exciting! There is room for engineers, like you, to make a fantastic impact in this space.

Testing software is the practice of proving that something works, that it doesn't fail, and that those conditions remain true in a variety of edge cases. So if we apply the same paradigm to testing infrastructure, our goals for testing are as follows:

1. Prove the infrastructure works as intended.
2. Prove the infrastructure isn't failing.
3. Prove that both conditions are true in a variety of edge cases.

Measuring whether or not infrastructure is working requires us to first define what working means. By now you should feel comfortable with the idea of representing infrastructure and engineering applications to manage the representation.

A user defining an infrastructure API should spend time focusing on creating a sane API that creates working infrastructure. For instance, it would be foolish to have an API that defines virtual machines, and no network information to run them in. You

should create useful abstractions in the API and then use ideas laid out in Chapters 3 and 4 to make sure the API creates the correct infrastructure components.

We have already begun to develop a mental model of what a sanity check of our infrastructure should look like just by defining the API. This means we can flip the logic and imagine the opposite case, which would be everything excluding what is in the original mental model.

The practice of defining basic sanity tests for infrastructure is a worthwhile endeavor. So the first step in testing infrastructure is to prove your infrastructure exists as intended and that nothing exists contrary to the original intent.

In this chapter we explore infrastructure testing and lay the groundwork for a new world order of testing tools.

What Are We Testing?

Before we can begin writing a line of code, we must first identify our concerns that will need to be tested.



Test-driven development is a common practice in which testing comes first. The tests are written to demonstrate the concerns of testing and will implicitly fail on day one. The development cycle is aimed at making the tests pass; that is, the software is developed to address the requirements defined in each test. This is a powerful practice that helps software stay focused and helps engineers gain confidence in their software and thus their tests.

This is a philosophical question that can be answered in a number of ways. Having a good idea of what we need to prove true and not false is imperative to building trustworthy infrastructure. For instance, if there is a business concern that relies on infrastructure being in place, it should be tested. More importantly, many businesses rely on infrastructure not only being in place, but also fixing itself if something goes wrong.

Identifying the problem space that your infrastructure needs to fill represents the first round of testing that will need to be written.

Planning for the future is another important aspect of infrastructure testing but should be taken lightly. There is a thin line between being sufficiently forward-looking and overengineering. If in doubt, stick with the bare minimum amount of testing logic.

After we have a solid understanding of what we need to demonstrate with our tests, we can look at implementing a testing suite.

Writing Testable Code

The rules of the reconciler pattern are not only aimed at creating a clean infrastructure application, but also explicitly designed to encourage testable infrastructure code.

This means that in every major step in our application we always fall back on creating a new object of the same type, which means every major component of our underlying system will speak the same input and output. Using the same inputs and outputs makes it easier to programmatically test small components of your software. Tests will ensure your components are working as expected.

However, there are many other valuable lessons to abide by while writing infrastructure-testing code. We will take a look at a concrete example of testing infrastructure in a hypothetical scenario. As we walk through the scenario, you will learn the lessons on testing infrastructure code.

We will also call out a handful of rules that engineers can abide by as they begin writing testable infrastructure code.

Validation

Take a very basic infrastructure definition such as [Example 6-1](#).

Example 6-1. infrastructure.json

```
{
  "virtualMachines": [{
    "name": "my-vm",
    "size": "large",
    "localIp": "192.168.1.111",
    "subnet": "my-subnet"
  }],
  "subnets": [{
    "name": "my-subnet",
    "cidr": "10.0.100.0/24"
  }]
}
```

It should be obvious what the data is aiming to accomplish: ensure a single virtual machine called `my-vm` of an arbitrary size `large` with an IP address of `192.168.1.111`. The data also alludes to ensuring a subnet with the name `my-subnet` that will house the virtual machine `my-vm`.

Hopefully you caught what is wrong with this data. The virtual machine has been given an IP address that falls outside of the available CIDR range for the subnet.

Running this particular data against an application should result in a failure, as the virtual machine will be effectively useless on the network. If our application was built to blindly allow any data to be deployed, we would create infrastructure that wouldn't work. While we should write a test to ensure that the new virtual machine is able to route on the network, there is something else we can do to help harden our application and make testing much easier.

Before our application entertains the idea of processing the input, we could first attempt to validate the input. This is a common practice in software engineering.

Imagine if, instead of blindly deploying this infrastructure, we first attempted to validate the input. At runtime our application could easily detect that the IP address for the virtual machine will not work in the subnet that the virtual machine is attached to. This would prevent the input from ever reaching our infrastructure environment. With the knowledge that the application will intentionally reject invalid infrastructure representation, we can write happy and sad tests to ensure this behavior is achieved.



A *happy test* is a test that exercises the positive case of a condition. In other words, it is a test that sends a valid API object to the application and ensures the application accepts the valid input. A *sad test* is a test that exercises the opposite case, or the negative case of a condition. **Example 6-1** is an example of a sad test that sends an invalid API object to the application and ensures the application rejects the invalid input.

This new pattern makes testing infrastructure very quick, and usually inexpensive. An engineer can begin to develop a large arsenal of happy and sad tests for even the most bizarre application input. Furthermore, the testing collection can grow over time; in the unfortunate scenario where an impostor API object trickles through to the environment, an engineer can quickly add a test to prevent it from happening again.

Input validation is one of the most basic things to test. By writing simple validation into our application that checks for sane values, we can begin to filter out the input of our program. This also gives us an avenue in which it is easy to define meaningful errors and return an error quickly.

Validation offers confidence without making you wait for infrastructure to be mutated. This creates a faster feedback loop for engineers developing against the API.

Entering Your Codebase

It is important that we continue to cater to ourselves and write code that is easily testable. An easy mistake that can be costly downstream is to engineer an application around proprietary input. Proprietary input is input that is relevant at only one point

in the program, and the only way to get the desired input is to linearly execute the program. Writing code linearly in this fashion makes sense to the human brain but is one of the hardest patterns to effectively test, especially when it comes to testing infrastructure.

An example of getting into trouble with proprietary input is as follows:

1. A call to function `DoSomething()` returns `Something{}`.
2. `Something{}` is passed to the function `NextStep(Something)` and `SomethingElse{}` is returned.
3. `SomethingElse{}` is passed to the function `FinalStep(SomethingElse)` and `true` or `false` is returned.

The problem here is that in order to test the `FinalStep()` function we first need to traverse steps 1 and 2. In the case of testing, this introduces complexity and more points of failure; it very well might not even work within the context of a test execution.

A more elegant solution would be to structure the code in such a way that `FinalStep()` could be called on the same data structure the rest of the program is using:

1. Code initializes `GreatSomething{}`, which implements the method `GreatSomething.DoSomething()`.
2. `GreatSomething{}` implements the method `GreatSomething.NextStep()`.
3. `GreatSomething{}` implements the method `GreatSomething.FinalStep()`.

From a testing perspective, we can populate `GreatSomething{}` for any step we wish to test, and call the methods accordingly. The methods in this example are now responsible for acting on the memory defined in the object they are extending. This is different than the last approach, where ad hoc in-memory structures were passed into each function.

This is a much more elegant design, since a test engineer can synthesize the memory for any step along the way easily and will only ever have to concern themselves with learning one representation of the same data. This is much more modular, and we can home in on any failures if they are detected quickly.

As you begin to write the software that makes up your application, remember that you will need to jump into your codebase at many points during the traditional runtime timeline. Structuring your code to make it easy to enter the codebase at any point, and therefore test the system internally, is critical. In situations like this, you can be your own best friend or worst enemy.

Self-Awareness

Tell me how you measure me, and I will tell you how I will behave.

—Eliyahu M. Goldratt

Pay attention to your confidence levels while writing your code and writing your tests. Self-awareness is one of the most important parts of software engineering and is frequently one of the most overlooked.

The ultimate goal of testing is to increase confidence in your application. In the realm of infrastructure, our goal is to increase our confidence that our infrastructure does what we want it to do.

With that being said, there is no right or wrong approach to testing infrastructure. It's easy to get caught up in metrics like code coverage or unit testing every function in your application. But these introduce false confidence.



Code coverage is the act of programmatically measuring how much of your codebase is being exercised by tests. This metric can be useful as a primitive datapoint, but it is critical to understand that even a codebase with 100 percent coverage could still be subject to an extreme outage.

If you measure your tests by code coverage, then engineers will write code that can more easily be covered by tests instead of code that is more appropriate for the task. Dan Ariely sums up human behavior this way in his article “[You Are What You Measure](#)” for the *Harvard Business Review*:

Human beings adjust behavior based on the metrics they're held against. Anything you measure will impel a person to optimize his score on that metric. What you measure is what you'll get. Period.

The only metric we should ever be measuring is our confidence that our infrastructure is working as intended and that we can demonstrate that.

Measuring confidence can be close to impossible. But there are ways that engineers have pulled meaningful data sets out of psychological and emotional spaces.

By asking ourselves questions like the following, we can record our answers over time:

- Am I worried this won't work?
- Can I be certain this will do what I think it will do?
- What happens if somebody changes this file?

One of the most powerful techniques in pulling data out of opinionated questions is to compare levels from previous experiences. For example, an engineer could make a statement like the following, and the rest of the team very quickly understands what they are trying to relay:

I am significantly more worried about this code breaking than I was worried about the release last quarter.

Now, based on the team's prior experience, we can begin to develop a scale for our confidence levels, with the 0 end of the scale being an experience when the team had virtually no confidence, and the upper end of the scale being a time they felt extremely confident. As we understand what worries us about our application, understanding what needs to be tested to increase confidence is simple.

Types of Tests

Understanding the types of tests, and how they are intended to be used, will help an engineer increase the confidence of their infrastructure applications. These tests are not necessary to write, and there is no right or wrong approach to using them. The only concern is that we trust our application to do what we want it to do.

Infrastructure Assertions

In software engineering, a powerful concept is the *assertion*, which is a way of forcefully determining if a condition is true. There are many successful frameworks that have been developed that use assertions to test software. An assertion is a tiny function that will test if a condition is true. These functions can be used in various testing scenarios to demonstrate that concepts are working, and hopefully introduce confidence.



Throughout the rest of the chapter we will be referring to infrastructure assertions. You will need to have a basic understanding of what these assertions look like, and what they hope to accomplish. You will also need a basic understanding of the Go programming language to fully realize what these assertions are doing.

There is a need in the infrastructure space for asserting that our infrastructure works. Building out a library of these assertion functions would be a worthwhile exercise for your project. The open source community could also benefit from having this toolkit to test infrastructure.

Example 6-2 shows an assertion pattern in the Go programming language. Let's imagine we would like to test if a virtual machine can resolve public hostnames and then route to them.

Example 6-2. assertNetwork.go

```
type VirtualMachine struct {
    localIp string
}

func (v *VirtualMachine) AssertResolvesHostname(
    hostname string,
    expectedIp string,
    message string) error {
    // Logic to query DNS for a hostname,
    // and compare to the expectedIp
    return nil
}

func (v *VirtualMachine) AssertRouteable(
    hostname string, r
    port int,
    message string) error {
    // Logic to verify the virtualMachine can route
    // to a hostname on a specific port
    return nil
}

func (v *VirtualMachine) Connect() error {
    // Logic to connect to the virtual machine to run the assertions
    return nil
}

func (v *VirtualMachine) Close() error {
    // Logic to close the connection to the virtual machine
    return nil
}
```

In this example we stub out two assertions as methods on the `VirtualMachine{}` struct. The method signatures are what we will be focusing on in this demonstration.

The first method, `AssertResolvesHostname()`, demonstrates a method that would be used to check if a given hostname resolves to an expected IP address. The second method, `AssertRouteable()`, demonstrates a method that would be used to check if a given hostname was routeable on a specific port.

Notice how the `VirtualMachine{}` struct has the member `localIp` defined. Also notice how the `VirtualMachine{}` struct has a `Connect()` function, as well as a `Close()` function. This is so that the assertion framework could run these assertions from within the context of the virtual machine. The test could run from a system outside the infrastructure environment and then connect to the virtual machine within the environment to run infrastructure assertions.

In [Example 6-3](#), we demonstrate how this would look and feel to an engineer writing tests on their local system using a Go test.

Example 6-3. network_test.go

```
func TestVm(t *testing.T) {
    vm := VirtualMachine{
        localIp: "10.0.0.17",
    }
    if err := vm.Connect(); err != nil {
        t.Fatalf("Unable to connect to VM: %v", err)
    }
    defer vm.Close()
    if err := vm.AssertResolvesHostname("google.com", "*",
        "google.com should resolve to any IP"); err != nil {
        t.Fatalf("Unable to resolve hostname: %v", err)
    }
    if err := vm.AssertRouteable("google.com", 443,
        "google.com should be routable on port 443"); err != nil {
        t.Fatalf("Unable to route to hostname: %v", err)
    }
}
```

The example uses the built-in testing standards in the Go programming language, meaning that this function will be executed as part of a normal `go test` run against the Go tests in your application. The testing framework will test all files with a name that ends in `_test.go` and test all functions with a signature name that starts with `TestXxx`. The framework will also pass in the `*testing.T` pointer to each of the functions defined in that way.

This simple test will use the assertion library we defined earlier to complete a few steps:

1. Attempt to connect to a virtual machine that should be reachable on 10.0.0.17.
2. From the virtual machine, attempt to assert that the virtual machine can resolve `google.com` and that it returns some IP address.
3. From the virtual machine, attempt to assert that the virtual machine can route to `google.com` on port 443.
4. Close the connection to the virtual machine.

This is an extremely powerful program. It introduces a high level of confidence that our infrastructure is working as intended. It also introduces an elegant palette for engineers to define tests without having to worry about how they will be run.

The open source community is in desperate need of infrastructure testing frameworks like this. Having a standardized and reliable way of defining infrastructure tests would be a welcome addition to an engineer's toolbox.

Integration Testing

Integration tests are also known as end-to-end (e2e) tests. These are long-running tests that exercise the system in the way it is intended to be used in production. These are the most valuable tests in demonstrating reliability and thus increasing confidence.

Writing an integration testing suite can be fun and rewarding. In the case of integration testing an infrastructure management application, the tests would perform a sweep of the life cycle of infrastructure.

A simple example of a linear integration testing suite would be as follows:

1. Define a commonly used infrastructure API.
2. Save the data to the application's data store.
3. Run the application, and create infrastructure.
4. Run a series of assertions against the infrastructure.
5. Remove the API data from the application's data store.
6. Ensure the infrastructure was successfully destroyed.

At any step along the way, the test might fail, and the testing suite should clean up whatever infrastructure it has mutated. This is one of the many reasons it is so important to test that destroying infrastructure works as expected.

The test gives us confidence that the application will create and destroy infrastructure as intended and that it works as expected. We can increase the number of assertions that are run in step 4 over time and continue to harden our suite.

The integration testing harness is potentially the most powerful context in which we can test our infrastructure. Without the integration testing harness, running smaller tests like unit tests wouldn't offer very much value.

Unit Testing

Unit testing is a fundamental part of testing a system and exercising its components individually. The responsibility of a unit test is intended to be small and discreet. Unit testing is a common practice in software engineering, and thus will be a part of infrastructure engineering.

In the case of writing infrastructure tests, testing one component of the system is difficult. Most components of infrastructure are built on top of each other. Testing

that software mutates infrastructure accordingly usually requires mutating infrastructure to test and see if it worked. This process will usually involve a large portion of a system.

But that does not mean it's impossible to write unit tests for an infrastructure management system. In fact, most of the assertions defined in the previous example are technically unit tests! Unit tests test only one small component, but they can be extremely helpful when used within the context of a larger integration testing system.

Unit tests are encouraged while you're testing infrastructure, but remember that the *context* in which they run usually requires a fair amount of overhead. This overhead usually comes in the form of integration testing. Combining the small and discreet checks of unit testing with the larger, overarching patterns of integration testing gives infrastructure engineers a high level of confidence that their infrastructure is working as intended.

Mock Testing

In software engineering, a common practice to synthesize systems is mock testing. In mock testing, an engineer writes or uses software that is designed to spoof, or fake, a system.

A simple example would be taking an SDK that is designed to talk to an API and running it in “mock” mode. The SDK doesn't send any data to the API, but rather synthesizes what the SDK *thinks* the API should do in various situations.

The responsibility of ensuring that the mock software accurately reflects the system that it is synthesizing lies in the hands of the engineers developing the mock software. In some cases, the mock software is developed by the same engineers that develop the system it is mocking.

While there might be some mock tools that are kept up to date and are more stable than others, there is one universal truth in using mock systems to synthesize the infrastructure you plan to test: fake systems give you fake confidence.

Now, this rule may seem harsh. But it is intended to encourage engineers to not take the easy way out, and to go through the practice of building a real integration suite to run their tests. While mock systems can be powerful, relying on them for the core of your infrastructure testing (and thus your confidence) is highly risky.

Most public cloud providers enforce quota limits for their resources. Imagine a test that was interacting with a system that had these hard limits on resources. The mock system might do its best to put a cap on resources—but without auditing the real system at runtime, the mock system would have no way of determining if your infrastructure would actually be deployed. In this case, your mock test would succeed. However, when the code is run in the real environment, it would break.

This is just one example of many that demonstrates why mutating real infrastructure and sending real network packets is far more reliable than using a mock system. Remember, the goal of testing is to increase confidence that your infrastructure will work as intended in a real environment.

This is not to say that all mock testing is bad. It is very important to understand the difference between mocking your infrastructure that you are testing and mocking another piece of the system for convenience.

The engineer will need to decide when is and isn't appropriate to use a mock system. We just caution engineers from developing too much confidence in these systems.

Chaos Testing

Chaos testing is probably the most exciting avenue of testing infrastructure that we will cover in this book. It's testing to demonstrate that unpredictable events in infrastructure can occur, without jeopardizing the stability of the infrastructure. We do this demonstration by intentionally breaking and disrupting infrastructure, and measuring how the system responds to the catastrophe. As with all our tests, we will approach this as an infrastructure engineer.

We will be writing software that is designed to break production systems in unexpected ways. Part of building confidence in systems is understanding how and why they break.

Building Confidence at Google

One example of learning how systems break can be seen in Google's DiRT (Disaster Recovery Training) program. The program is intended to help Google's site reliability engineers stay familiar with the systems they support. In *Site Reliability Engineering*, they explain that the purpose of the DiRT program is because "being out of touch with production for long periods of time can lead to confidence issues, both in terms of overconfidence and underconfidence, while knowledge gaps are discovered only when an incident occurs."

Unfortunately, it will not do the engineering team much good to just break production without having systems in place to measure the impact and recover from the catastrophe. Again, we will call on the infrastructure assertions defined earlier in the chapter. The tiny, single responsibility functions offer fantastic datapoints for measuring systems' stability over time.

Measuring chaos

Let's look at the `AssertRouteable()` function from [Example 6-3](#) again. Imagine that we have a service that will connect to a virtual machine and attempt to keep the connection open for eternity. Every second the service will call the `AssertRouteable()` function and log the result. The data from this service is an accurate representation of the virtual machine's ability to route on their network. As long as the virtual machine can route, the data would yield a straight, unchanged line on a graph as in [Figure 6-1](#).

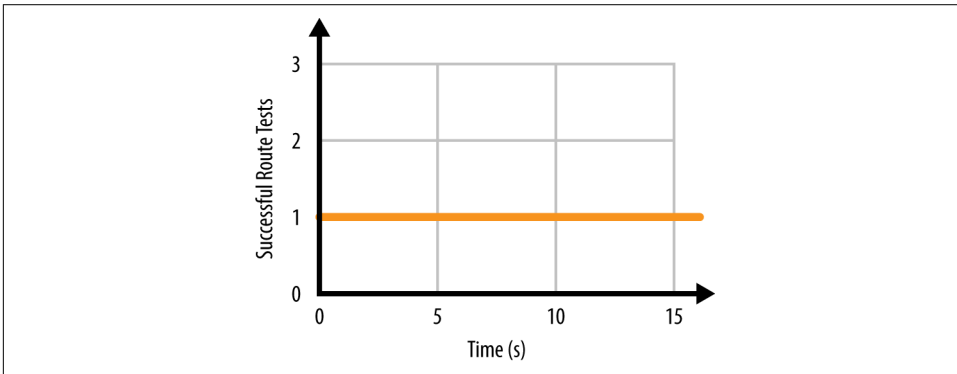


Figure 6-1. Graph of `AssertRouteable` tests over time

If at any time the connection broke, or the virtual machine was no longer able to route, the graph data would shift, and we would see a change in the line on the graph. Over time, as the infrastructure repaired itself, the line on the graph would again stabilize, as seen in [Figure 6-2](#).

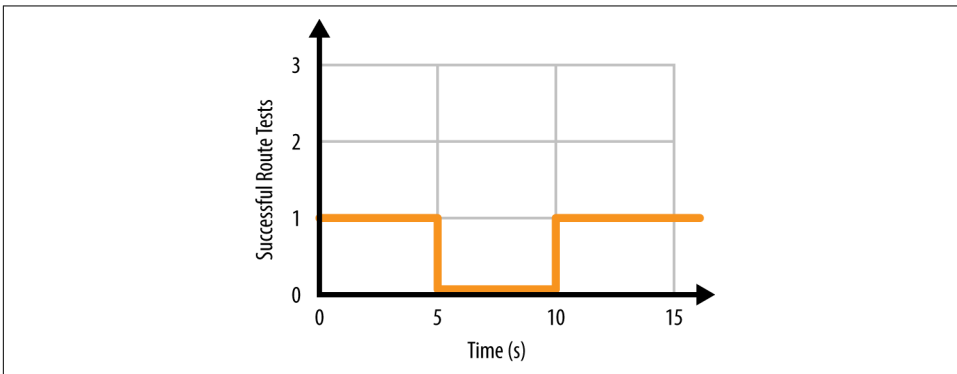


Figure 6-2. Failed and repaired `AssertRouteable` tests over time

The important dimension to consider here is time. Measuring chaos implicitly comes with the measurement of chaos over time.

We can expand our measurements very quickly. Imagine that the service that called `AssertRouteable()` was now calling a set of 100 infrastructure assertions on the virtual machine. Also imagine that we had 100 virtual machines that we were measuring.

This would yield roughly 1.0×10^4 assertions per second against our infrastructure. This enormous amount of data coming out of our infrastructure assertions allows us to create powerful graphs and representations of our infrastructure. Recording the data in a queryable format also allows for advanced after-chaos investigation.

With measuring chaos, it is important to have reliable measuring tools and services. It is also important to store the data from the services in meaningful ways so it can be referenced later. Storing the data in a log aggregator or another easily indexed data store is highly encouraged.

The chaos of a system is inversely related to the reliability of the system. Therefore, it is a direct representation of the stability of the infrastructure we are evaluating. That means it is valuable to have the information graphed over time for analysis when things break or when introducing change to know if it decreased the stability.

Introducing chaos

Introducing chaos into a system is another way of saying “intentionally breaking a system.” We want to synthesize unexpected permutations of infrastructure issues that we might see in the wild. If we don’t intentionally inject chaos, the cloud provider, internet, or some system will do it for us.

Netflix’s Simian Army

Netflix introduced what it calls the Simian Army to cause chaos in its systems. Monkeys, apes, and other animals in the simian family are each responsible for causing chaos in different ways. Netflix explains how one of these tools, Chaos Monkey, works:

This was our philosophy when we built Chaos Monkey, a tool that randomly disables our production instances to make sure we can survive this common type of failure without any customer impact. The name comes from the idea of unleashing a wild monkey with a weapon in your data center (or cloud region) to randomly shoot down instances and chew through cables—all the while we continue serving our customers without interruption. By running Chaos Monkey in the middle of a business day, in a carefully monitored environment with engineers standing by to address any problems, we can still learn the lessons about the weaknesses of our system, and build automatic recovery mechanisms to deal with them. So next time an instance fails at 3 a.m. on a Sunday, we won’t even notice.

In terms of cloud native infrastructure, the monkeys are great examples of infrastructure as software and utilizing the reconciler pattern. A major difference is that they are designed to destroy infrastructure in unexpected ways instead of creating and managing infrastructure predictably.

At this point you should have an infrastructure management application ready to use, or at least one in mind. The same infrastructure management application used to deploy, manage, and stabilize infrastructure can also be used to introduce chaos.

Imagine two very similar deployments.

The first, [Example 6-4](#), represents working (or happy) infrastructure.

Example 6-4. infrastructure_happy.json

```
{
  "virtualMachines": [{
    "name": "my-vm",
    "size": "large",
    "localIp": "10.0.0.17",
    "subnet": "my-subnet"
  }],
  "subnets": [{
    "name": "my-subnet",
    "cidr": "10.0.100.0/24"
  }]
}
```

We can deploy this infrastructure using the means set up in the environment. This infrastructure should be deployed, running, and stable. Just as before, it's important to record your infrastructure tests over time; [Figure 6-3](#) is an example of this. Ideally, the amount of tests you run should increase over time.

We decide we want to introduce chaos. So we create a copy of the original infrastructure management application, although this time we take a far more sinister approach to deploying infrastructure. We take advantage of our deployment tool's ability to audit infrastructure and make changes to infrastructure that already exists.

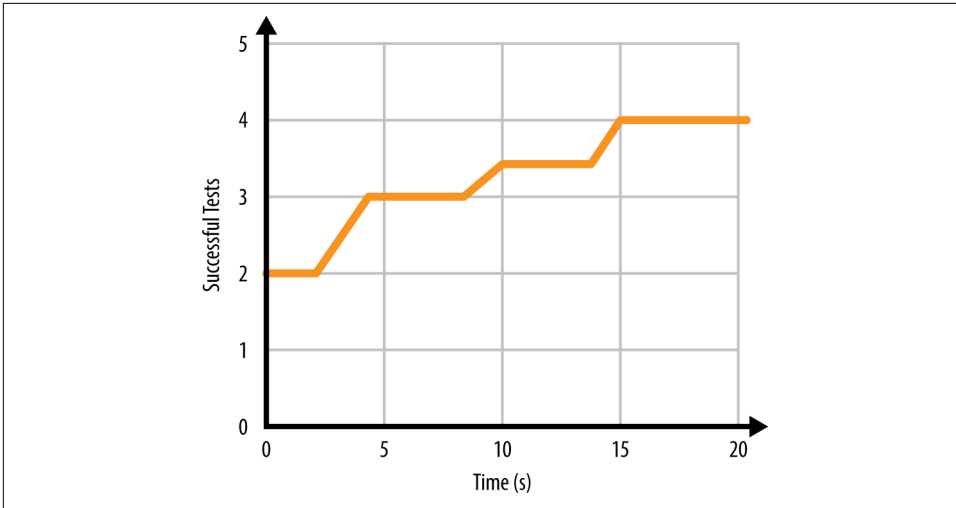


Figure 6-3. Successful tests measured over time

The second deployment would represent intentionally faulty infrastructure and still use the same identifiers (names) as the original infrastructure. The infrastructure management tool will detect existing infrastructure and change it. In the second example (Example 6-5), we change the virtual machine size to `small` and intentionally assign the virtual machine a static IP address `192.168.1.111` that is outside the cidr range `10.0.100.0/24`.

We know the workload on the virtual machine will not run on a small virtual machine, and we know the virtual machine will not be able to route on the network. This is the chaos we will be introducing.

Example 6-5. `infrastructure_sad.json`

```
{
  "virtualMachines": [{
    "name": "my-vm",
    "size": "small",
    "localIp": "192.168.1.111",
    "subnet": "my-subnet"
  }],
  "subnets": [{
    "name": "my-subnet",
    "cidr": "10.0.100.0/24"
  }]
}
```

As the second infrastructure management application silently makes changes to the infrastructure, we can expect to see things break. The data on our graphs will begin fluctuating, as seen in [Figure 6-4](#).

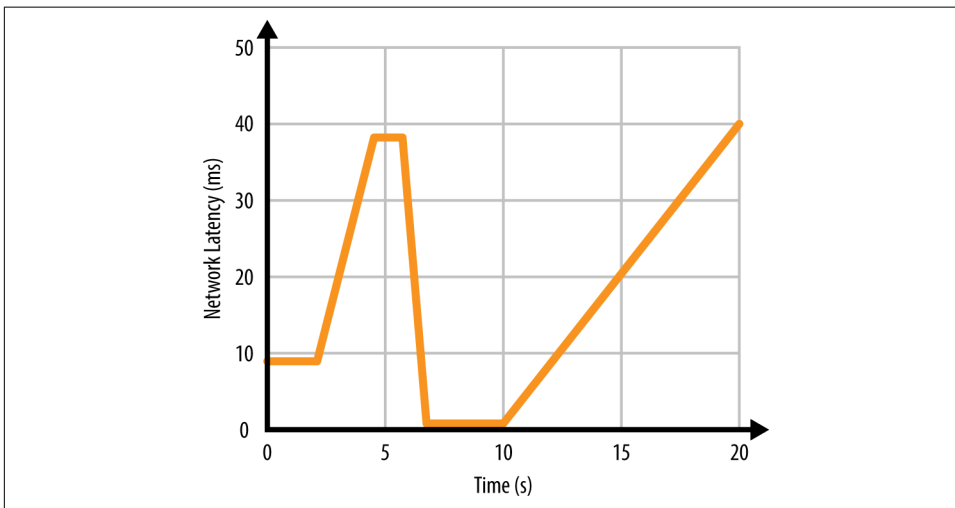


Figure 6-4. Graph with network failures

Any applications on the virtual machine that have been altered should slowly fail, if they don't break entirely. The virtual machine's memory and CPU are now overloaded. The shell is unable to fork new processes. The load average is well above 20. The system is approaching deadlock, and we can't even access the virtual machine to see what is wrong because nothing can route to the impostor IP address.

As expected, the initial system will detect that something has changed in the underlying infrastructure and will reconcile it accordingly. It is important that the impostor system be taken offline, or else there is a risk of never-ending reconciliation between the two systems, which will compete to rectify the infrastructure in the way they were instructed to do so.

The beauty of this approach to introducing chaos is that we didn't have to develop any extra tooling or spend any engineering hours writing chaos frameworks. We abused the original infrastructure management tool in a clever way to introduce a catastrophe.

Of course that may not always be a perfect solution. Unlike your production infrastructure applications, your chaos applications should have constraints to make sure they are beneficial. Some common constraints are the ability to exclude certain systems based on tags or metadata, not run chaos tests during off hours, and limit chaos to a certain percentage or type of system.

The burden of introducing random chaos now lies in an infrastructure engineer's ability to randomize the workflow we just explored over time. Of course, the infrastructure engineer will also need to ensure that the data gathered from their experiment is offered in a digestible format.

Monitoring Infrastructure

Along with testing infrastructure, we cannot forget to monitor what is actively running. Tests and familiar failure patterns can go a long way to give you confidence in your infrastructure, but it is impossible to test every way a system can fail.

Having monitoring that can detect anomalies not identified during testing and perform the correct actions is crucially important. With active monitoring of the live infrastructure, we can also increase our confidence that when something happens that is not recognized as “normal,” we will be alerted. Knowing when and how anomalies should alert a human is a topic of much debate.

There are lots of good resources that are emerging from the practice of monitoring infrastructure in a cloud native environment. We will not address the topics here, but you should start by reading *Monitoring Distributed Systems: Case Studies from Google's SRE Teams* (O'Reilly) by Rob Ewaschuk and watching videos from the [Monitorama conference](#). Both are available for free online.

Whatever monitoring solution you implement, remember the cloud native approach to creating your monitoring rules. Rules should be declarative and stored as code. Monitoring rules should live with your application code and be available in a self-service manner. Don't overcompensate for monitoring when testing and telemetry will likely fulfill most of your needs.

Conclusion

Testing needs to introduce confidence and trust in the infrastructure so we gain confidence and trust in the applications we are supporting. If a testing suite does not offer confidence, its value should be in question. Try to remember that the tools and patterns suggested in this chapter are starting points and are designed to excite and engage an engineer working in this space. Regardless of the types of tests, or the framework that is used to run them, the most important takeaway is that an engineer can begin to trust their systems. As engineers, we typically gain trust by observing hands-on demonstrations that prove something is working as expected.

Furthermore, experimenting in production is not only OK, it is encouraged. You will want to make sure the environment was built for such experimentation and proper tracking is implemented so the testing does not go to waste!

Measuring reality is an important part of infrastructure development and testing. Being able to encapsulate reality from both an engineering perspective and an operating perspective is an important part of exercising infrastructure and thus gaining confidence that it works as intended.

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 functionality 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 (see [Appendix A](#)). 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, as explained in [Chapter 5](#). 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 7-1](#). APM adds tools to your applications for reporting metrics and faults to a collection service.

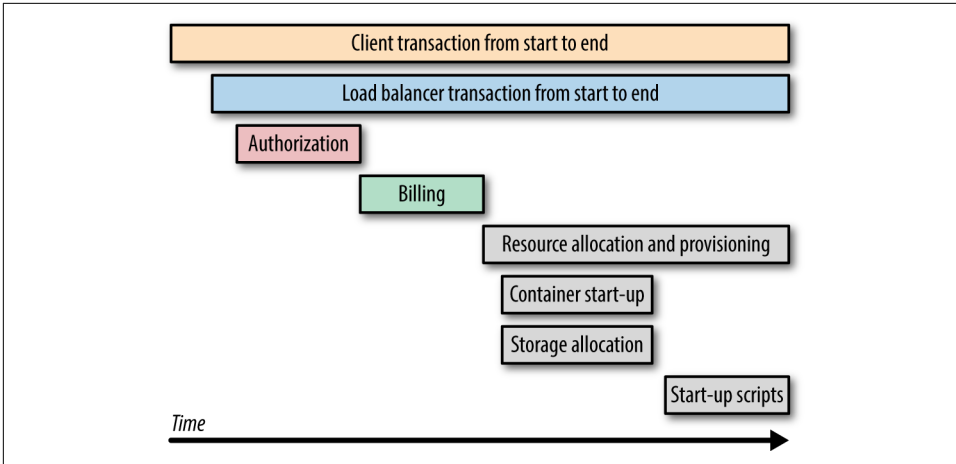


Figure 7-1. OpenTracing visualization

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.

Securing Applications

We've already discussed that infrastructure should only be provisioned from cloud native applications. We know that the infrastructure is also responsible for running those same applications.

Running infrastructure provisioned and controlled by applications allows us to scale more easily. We scale our infrastructure by learning how to scale applications. We also secure our infrastructure by learning how to secure applications.

In a dynamic environment, we have shown that humans cannot scale to manage the complexity. Why would we think they can scale to handle the policy and security?

This means, just as we had to create the applications that enforce infrastructure state through the reconciler pattern, we need to create applications that enforce security policy. Before we create the applications to enforce policy, we need to write our policy in a machine-parsible format.

Policy as Code

Policy is hard to put into code because it does not have cleanly defined technical implementations. It deals more with *how* business gets done more than *what* runs business.

Both the how and the what change frequently, but the how is much more opinionated and not easily abstracted. It also is organization specific and will likely need to know specific details about the communication structure of the people who create the infrastructure.

Policy needs to apply to multiple stages of the application life cycle. As we discussed in [Chapter 7](#), applications generally have three stages; deploy, run, and retire.

The deploy stage will have policy applied before the application and infrastructure changes go out. This will include deployment rules and conformity tests. The run stage will include ongoing compliance and enforcement of access controls and segregation. The retire stage is important to make sure no services are left behind in unpatched or unmaintained states.

In each of these stages, you need to break down policies into a definitive, actionable implementation. Vague policy cannot be enforced. You will need to put the implementation in code and then create the application, or use an existing application, to enforce the policy rules.

Once you have policy as code, you should treat it as code. Policy changes should be treated as application changes and tracked in version control.

The same policy that gates your application deployments should also apply to your new policy deployments. The more components of infrastructure you can have tracked and deployed with the same tools as deploying applications, the easier it will be to understand what is running and how changes can impact the system.

A great benefit about policy as code is that you can easily add or remove policies, and they are tracked, so there's a record of who did it, when they did it, as well as comments around the commits and pull request. Because the policy exists as code, you can now write tests for your policies! If you want to validate that a policy will do the right thing, you can use the testing practices from [Chapter 5](#).

Let's look more closely at applying policy to application life cycles.

Deployment Gating

Deployment gating makes sure *how* applications get deployed conform to the business rules. This means you will need to build a deployment pipeline and not allow direct-to-production deployments from users' machines.

You will need to centralize control before enforcing centralized policy, but you should start small and prove that the solution works before implementing it everywhere. The benefits of a deployment pipeline go well beyond policy enforcement and should be standard in any organization with more than a handful of developers.

Here are some examples of policy:

- Deployments can happen only after all tests have passed.¹

¹ Not to mention they should require tests in the first place.

- New applications require a senior developer to review the changes and comment on the pull request.
- Production artifact pushes can happen only from the deployment pipeline.



Gating is not supposed to enforce the running state or API requests to applications. The application should know how infrastructure components are provisioned and policy is applied to those components through compliance and auditing and not during application deployment.

An example of a deployment gating policy is if your organization doesn't allow code deploys past 3 p.m. on a Friday without manager approval.

This one is pretty easy to put into code. [Figure 8-1](#) is a very simplified diagram to represent the policy.

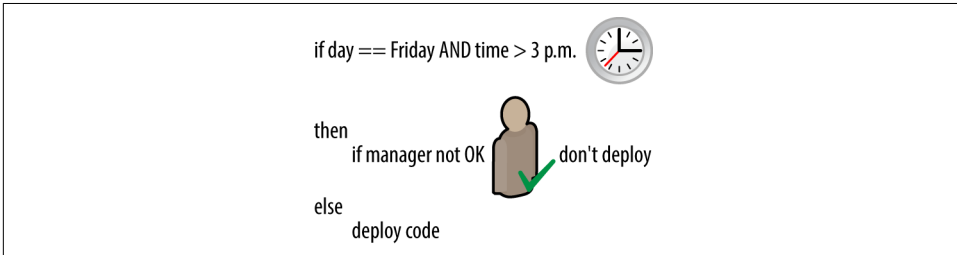


Figure 8-1. Deployment policy

You can see the policy simply checks the time and day of the week when a deployment is allowed to be deployed. If it is Friday and after 3 p.m., then the policy checks for a manager OK.

The policy can get the OK notification via a verified email sent by the manager, an authenticated API call, or various other means.² It is up to the policy to decide what the preferred communication method is and how long to wait for approval.

This simple example can be expanded with lots of different options, but it's important to make sure the policy is not up to humans to parse and enforce. Human interpretation differs, and unclear policies often don't get enforced at all.

By making sure the policy will block new deployments, we can save ourselves a lot of work in troubleshooting the state of the production environment. Having a chain of events that can be verified in software goes a long way in understanding your

² Authentication is crucial when enforcing policy.

systems. Version control and continuous deployment pipelines can verify code; using policy and process as code can verify how and when software is deployed.

Beyond just making sure the right thing gets deployed via company policy, we should also make it easy to deploy the supported thing with templates and enforce them with conformity tests.

Conformity Testing

In any infrastructure you need to provide a recommended way to create specific types of applications. These recommendations become building blocks for users to consume and piece together depending on their needs.

It's important that the recommendations are composable but not too small. They need to be understandable in their intended functionality and self-service. We've already recommended that cloud native applications be packaged as containers and consumed via an orchestrator; it is up to you to figure out what works best for your users and what components you want to provide.³



There are multiple ways you can provide templated infrastructure to users.

Some examples are to provide templated code such as [Jsonnet](#), or fully templated applications such as charts with [Helm](#).

You also could provide templates via your deployment pipeline. These can be Terraform modules or deployment-specific tools such as Spinnaker templates.

Creating deployment templates allows users to become consumers of the template, and as best practices evolve, the users automatically benefit.

The most critical aspect of infrastructure templates is to make it easy to do the right thing and hard to do the wrong thing. If you meet the customer's need, it will be much easier to get adopters.

However, the whole reason we need templated infrastructure is so we can enforce conformity tests. Conformity tests exist to make sure applications and infrastructure components adhere to the organization's standards.

³ If your template unit is IaaS components (e.g., VM and VPC), you need to read [Chapter 1](#) again to understand what cloud native infrastructure is.

Some examples of testing for infrastructure that does not adhere to standards are:

- Services or endpoints not in an automatic scaling group
- Applications not behind a load balancer
- Frontend tiers talking directly to databases

These standards are information about infrastructure that you can find by making a call to the cloud provider's API. Conformity tests should be run continually and enforce the architectural standards your company has adopted.

If infrastructure components or application architecture are found to violate the provided standards, they should be terminated as early as possible. The sooner you can codify your templates, the earlier you can check for applications that don't conform to your standards. It's important to address unsupported architecture early in the application's life so reliance on the architectural decisions can be minimized.

Conformity deals with how applications are built and maintaining operability. Compliance testing makes sure the application and infrastructure stay secure.

Compliance Testing

Compliance testing does not test architectural designs, but rather it focuses on the implementation of components to make sure they adhere to defined policies. The easiest policies to put in code are those that aid with security. Policies around organizational requirements (e.g., HIPAA) should also be defined as code and tested during compliance testing.

Some examples of compliance policies are:

- Object storage has limited user access and is not readable or writable by the public internet
- API endpoints all use HTTPS and have valid certificates
- VM instances (if you have any) do not have overly permissive firewall rules

While these policies do not make applications free from all exploits or bugs, compliance policies should minimize the scope of impact if an application does get exploited.

Netflix explains the purpose of its implementation of compliance testing via its Security Monkey in its blog post "[The Netflix Simian Army](#)":

Security Monkey is an extension of Conformity Monkey. It finds security violations or vulnerabilities, such as improperly configured AWS security groups, and terminates the offending instances. It also ensures that all our SSL and DRM certificates are valid and are not coming up for renewal.

Putting your policy into code and continually running it by watching the cloud provider API allows you to stay more secure, catch insecure settings immediately, and track the policy over time with version control systems. The model of continually testing your infrastructure against these policies also fits nicely with the reconciler pattern.

If you think of policy as a type of configuration that needs to be applied and enforced, then it can be much simpler to implement it. It's important to keep in mind that as your infrastructure and business needs change, so should your compliance policies.

Deployment testing watches applications before they are deployed to the infrastructure, and conformity and compliance testing both deal with running applications. The last application life cycle phase to make sure you have policy for is knowing when and how to retire applications and life cycle components.

Activity Testing

Conformity and compliance testing should remove applications and infrastructure that fail the defined policies. There should also be an application that cleans up old and unused infrastructure components. High-level usage patterns should be based on application telemetry data, but there are still other infrastructure components that are easily forgotten and need to be retired.

In a cloud environment, you can consume resources on demand, but it is all too easy to forget what you demanded. Without automatic cleanup of old or unused resources, you will end up with a big surprise in your usage bill or need to spend costly human hours to do manual auditing and cleanup.

Some examples of resources you should test for and automatically clean up include:

- Old disk snapshots
- Test environments
- Previous application versions

The application responsible for cleanup needs to do the right thing according to the policy by default and provide flexibility for exceptions to be specified by engineers.

As mentioned in [Chapter 7](#), Netflix has implemented what it calls its “Janitor Monkey,” whose implementation perfectly describes this needed pattern:

Janitor Monkey works in a process of “mark, notify, and delete”. When Janitor Monkey marks a resource as a cleanup candidate, it schedules a time to delete the resource. The delete time is specified in the rule that marks the resource.

Every resource is associated with an owner email, which can be specified as a tag on the resource or you can quickly extend Janitor Monkey to obtain the information from your internal system. The simplest way is using a default email address, e.g. your team's

email list for all the resources. You can configure a number of days for specifying when to let Janitor Monkey send notifications to the resource owner before the scheduled termination. By default the number is 3, which means that the owner will receive a notification 3 business days ahead of the termination date.

During the 3 day period, the resource owner can decide if the resource is OK to delete. In case a resource needs to be retained longer the owner can use a simple REST interface to flag the resource as not being cleaned by Janitor Monkey. The owner can always use another REST interface to remove the flag and Janitor Monkey will then be able to manage the resource again.

When Janitor Monkey sees a resource marked as a cleanup candidate and the scheduled termination time is already passed, it will delete the resource. The resource owner can also delete the resource manually if he/she wants to release the resource earlier to save cost. When the status of the resource changes which makes the resource not a cleanup candidate, e.g. a detached EBS volume is attached to an instance, Janitor Monkey will unmark the resource and no termination will happen.

Having an application that automatically cleans up your infrastructure will keep your complexity and costs lower. This testing implements the reconciler pattern to the last life cycle stage of our applications.

There are still other practices with infrastructure that are important to consider. Some of them apply to traditional infrastructure, but in a cloud native environment need to be handled differently.

Continually testing aspects of the infrastructure helps you to know you're adhering to your policies. When infrastructure comes and goes frequently, it is difficult to audit what changes may be responsible for an outage or to use historical data to predict future trends.

If you expect to get that information from billing statements or by extrapolating a current snapshot of the infrastructure, you will discover quickly that the information provided there is the wrong tool for the job. For tracking changes and predicting the future, we need to have auditing tools that can quickly provide us the information we need.

Auditing Infrastructure

Auditing cloud native infrastructure in this sense is not the same as auditing components in the reconciler pattern, nor is it the same as the testing frameworks discussed in [Chapter 6](#). Instead, when we talk about auditing, we mean a high-level overview of change and component relations within the infrastructure.

Keeping track of what existed in the infrastructure and how it relates to other components gives us important context when trying to understand the current state. When something breaks, the first question is almost always, "What changed?" Auditing

answers that question for us and can be leveraged to tell us what will be affected if we apply a change.

In traditional infrastructure, the configuration management database (CMDB) was often the source of truth for the current state of the infrastructure. However, CMDBs do not track historical versions of the infrastructure or asset relationships.

Cloud providers can give you CMDB replacements via their inventory APIs, but they may not be incentivized to show you historical trends or let you query on specific details you need for troubleshooting, such as hostnames.

A good cloud auditing tool will allow you to show the difference (“diff”) of your current infrastructure compared to yesterday’s or last week’s. It should be able to combine cloud provider data with other sources (e.g., container orchestrator) to allow you to query the infrastructure for data the cloud provider may not have, and ideally it can automatically build topographical representations of component relationships.

If your applications run entirely on a single platform (e.g., Kubernetes), it is much easier to gather topology information for resource dependencies. Another way to automatically visualize relationships is by unifying the layer at which relationships happen.

For services running in the cloud, relationships can be identified in the network communication between the services. There are many ways to identify the network traffic between services, but the important consideration with auditing is keeping historical track of the information. You need to be able to easily identify when relationships change just as easily as you identify when components come and go.⁴

Methods for Automatically Identifying Service Relationships

Tracking network traffic between services means you need a tool that is aware of the network protocols used for communication. You cannot simply depend on raw packet traffic flowing in or out of a service endpoint. You’ll need a way to tap into the flow of information to build the relationship model.

A popular way to inspect the network traffic and build a dependency graph is through network proxies.

Some implementation examples for network proxies are linkerd and envoy. Services are tooled to flow all traffic through these proxies, which are aware of protocols being used and other dependent services. Proxies also allow for other network resiliency patterns, as discussed in [Appendix B](#).

⁴ Changes include new dependencies as well as existing dependencies that are used significantly more or less.

Tracking topology in relation to time is a powerful auditing tool. Combined with infrastructure history it will make sure the “what changed?” question is easier to answer.

There is also an aspect of auditing that deals with building trust in the current state of the infrastructure. Applications gain trust through the testing tools described, but some aspects of infrastructure cannot have the same tests applied.

Building infrastructure with verifiable, reproducible components can provide a great deal of trust. This practice is known as immutable infrastructure.

Immutable Infrastructure

Immutable infrastructure is the practice of creating change through replacing rather than modifying. In other words, instead of running configuration management to apply a change on all of your servers, you build new servers and throw away the old ones.

Cloud environments greatly benefit from this method of creating change because the cost of deploying new VMs is often so low it is easier to do than to manage another system that can enforce configuration and keep instances running. Because servers are virtual (i.e., defined in software), you can apply the same practices used to build your applications as you can for building your server images.



Traditional infrastructure with physical servers has an exact opposite optimization as the cloud when it comes to system creation. Provisioning a physical server takes a long time, and there is great benefit in modifying existing operating systems rather than replace and throw away the old ones.

One of the problems with immutable infrastructure is building the chain of trust to create the golden image for deployment. Images, when built, should be verifiable (e.g., signing keys or image hash), and once deployed should not change.

Image creation also needs to be automated. One of the biggest issues with historical use of golden images is they often relied on humans to run through checklists in time-consuming creation processes.

Tools exist (e.g., Packer by Hashicorp) to automate the build process, and there is no reason to have old images. The automation also allows the old checklist to become a script that can be audited and version-controlled. Knowing when the provisioning tools change and by whom is another aspect of immutable infrastructure that builds trust.

When change occurs, and it should change frequently, you need a way to track what changed and why. Auditing will help identify what changed, and immutable infrastructure can help you track back to a Git commit or pull request.

Having a tracked history also greatly helps with recovering from failure. If you have a VM image that is known to work and one that does not, you can deploy the previous, working version just as quickly as you deployed the new, broken version.

Immutable infrastructure is not a requirement of cloud native infrastructure, but the environment greatly benefits from this style of infrastructure management.

Conclusion

Through the practices described in this chapter, you will be able to more easily control what runs in your infrastructure and track how it got there. Putting your policies in code will allow you to track changes, and you will not rely on humans to interpret the policy properly.

Auditing and immutable infrastructure give you much better information to make sure your systems are secure and help you recover from failure faster.

We will not discuss security in this book beyond the compliance testing requirements discussed earlier in this chapter, but you should keep up to date with security best practices for the technologies and cloud provider you use. One of the most important aspects to security is to apply “security in layers” in the entire stack.

In other words, just running an antivirus daemon on the host operating system is not enough to secure your applications. You’ll need to look at all layers of the infrastructure that you control and apply the security monitoring applicable to your needs.

All the testing described in this chapter should run in the same reconciler pattern described in [Chapter 4](#). Gathering information to know the current state, looking for changes based on a set of rules, and then making the changes all fit cloud native patterns.

If you can implement your policies as code, you will go beyond technical benefits of the cloud and realize business benefits to cloud native patterns.

Historical and topographical auditing may not seem like clear benefits, but will be very important as the infrastructure grows and the rate of change and application agility increases. Applying traditional methodologies to managing cloud infrastructure is not cloud native, and this chapter has shown you some of the benefits you should leverage and some of the challenges you will face.

Implementing Cloud Native Infrastructure

If you thought cloud native infrastructure was a product you could buy or a cloud provider where you could run your servers, we are sorry to disappoint. It is not something you can benefit from without adopting the practices and changing how you build and maintain infrastructure.

It impacts much more than just servers, network, and storage. It is about how engineers manage applications just as much as it is about embracing failure.

A culture built around cloud native practices is very different from traditional technology and engineering organizations. We are not experts in addressing organizational cultures or structures, but if you're looking to transform your organization, we suggest looking at values and lessons from implementing DevOps practices in high-performing organizations.

Some places to explore are [Netflix's culture deck](#), which promotes freedom and responsibility, and Amazon's two-pizza teams, which promote autonomous groups with low overhead. Cloud native applications need to have the same decoupled characteristics as the teams that built them. Conway's law describes it best: "organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations."¹

As we close out this book, we'd like to focus on what areas are the most important as you adopt cloud native practices. We'll also discuss some of the patterns in infrastructure that are predictors of change, so you know what to look for in the future.

¹ M.E. Conway, "Design of a separable transition-diagram compiler," *Communications of the ACM* 6, no. 7 (July 1963).

Where to Focus for Change

If you have existing infrastructure or traditional data centers, the transition to cloud native will not happen overnight. Trying to force a new way of managing infrastructure will likely fail if you have any sizable infrastructure footprint or more than two people managing it.

The main places of focus to adopt these patterns surprisingly have little to do with provisioning new servers or buying new software. To begin adopting cloud native infrastructure, it's important that you focus on these areas first:

- People
- Architecture
- Chaos
- Applications

Don't start changing your infrastructure until you have prepared these areas to run with the practices describe in this book.

People

The ability to learn faster than your competitors may be the only sustainable competitive advantage.

—Arie de Geus

As we discussed in [Chapter 2](#), people are the hardest part of implementing any change. There are many reasons for their resistance, and it is the responsibility of those asking for change to assist those it will affect.

Changes are easier when there is a driving incentive for the need to change. Being proactive to improve potential is a good motivator, but it can be hard to modify behavior without a sense of urgency.

Many reasons people are resistant to change come from fear. People like habits because they feel in control and avoid surprises.

In order to make a major transition in any technology successful, you need to work with people to minimize their fears. Give them a sense of ownership, and explain clear goals for the change. Make sure to highlight the similarities between the new and old technologies, especially when it comes to the role they will play after the change.²

² Giving them this book to read is also a good first step!

It is also important that people understand the change is not because of their failures with the old or existing systems. They need to understand that the requirements have changed and the environment is different, and you want them to be a part of the change because you respect what they have done and have confidence in what they can do.

They will need to learn new things, and for that to happen, failures are necessary, expected, and a sign of progress.

Encourage learning and experimentation, and reward people and systems that adapt to finding insights in data. One way you can do this by letting engineers explore new possibilities through freedoms such as “20 percent time.”³

An agile system has no benefits if it doesn’t change. A system that does not adapt and improve will not be able to meet the needs of a business that is changing and learning.

Once you are able to excite people for change, you should empower them with trust and freedom. Continually guide them to align their goals with the business needs, and give them responsibility for what they were hired to manage.

If people are ready to adopt the practices in this book, there are few limitations in making it happen. For a much more in-depth guide on creating change in an organization, we recommend reading *Leading Change* by John P. Kotter (Harvard Business Review Press).

Changing the culture of your environment requires a lot of effort and buy-in from the organization, as does changing the infrastructure where your applications run. The architecture you choose can have a significant impact on your ability to adopt cloud native patterns.

Architecture

Resilience, security, scalability, deployability, testability are architectural concerns.

—Jez Humble

When migrating applications to cloud native infrastructure, you need to consider how the application is managed and designed. For example, 12-factor applications, the predecessor to cloud native applications, benefit from running on a platform. They are architected for minimal manual management, frequent changes, and resiliency. Many traditional applications are architected to resist automation, infrequent

³ 20 percent time is the policy that engineers should spend up to 20% (8 hours) of their working hours on projects of their choice. Products such as Gmail, AdSense, and Post-it notes were created as a result of this practice.

upgrades, and failures. You should consider the application's architecture before moving it.

Individual application architecture is one concern, but you also need to consider how the application communicates with other services inside your infrastructure. Applications should communicate with protocols supported in cloud environments and through clearly defined interfaces. Keeping application scope small by adopting microservices can help keep interapplication interfaces defined and application deployment velocity up. However, adopting microservices will expose new problems, such as slower application communication and the need for distributed tracing and policy-controlled networking. Do not adopt microservices—and the problems they bring—without providing benefit in your infrastructure.

While you *can* fit just about any application to run in a container and be deployed with a container orchestrator, you will quickly doom your efforts if the first choice is to migrate all of your business-critical database servers.⁴

It would be a better idea to first identify applications that are close to having the characteristics outlined in [Chapter 1](#) and gain experience running them in a cloud native environment. Once you collectively gain experience and good practice around the simple applications, then you can decide what to move next.

Your servers and services are no different. Before you switch your infrastructure to be immutable, you should make sure it addresses your current problems and that you are aware of the new problems.

The most important architectural consideration in reliable systems is to strive for simple solutions. Software and systems naturally veer toward complexity, which will create instability. In a cloud, you release control of so many areas that it's important to retain simplicity in the areas you can still control.

Whether you move your on-premises infrastructure to the cloud or create new solutions there, make sure you do the availability math in [Chapter 1](#) and are prepared for the chaos.

⁴ Database engines such as MySQL and PostgreSQL have architectures that resist automation in many ways. You can run them in containers, but they require more expertise than applications designed as cloud native.

Chaos Management

Embrace failure and expect chaos.

—Andrew Spyker, Netflix

When you're building cloud native applications and infrastructure, the goal is to create the minimal viable product (MVP) and iterate.⁵ Trying to dictate what your customers want or how they will use your product may be useful for a while, but successful applications will adapt instead of predict.

Customers' needs change; applications need to change with them. You cannot plan and build a complete solution, because by the time the product is ready, the demands have changed.

Remaining agile and building on existing technology is important. Just as you import libraries for applications, you should consume IaaS and SaaS for infrastructure. The more you try to build yourself, the slower you will be able to provide value.

Whenever you release control of something, you risk the unexpected. If you've ever had an application break because an imported library was updated, you'll know what this feels like.

Your application was dependent on functionality provided by the library, which changed. Should you remove the library and write your own functionality so you can control it? The answer is almost always no. Instead, you update your application to use the new library, or you bundle the functioning older version of the library with your application to temporarily avoid breakage.

The same is true for infrastructure running on a public cloud. You no longer have control of the hardwired network, what RAID controllers are used, or what version of the hypervisor runs your VMs. All you have are APIs that can provision abstractions on top of the underlying technology.

You do not control when the underlying technology is changed. If the cloud provider deprecates all large memory instance types, you have no choice but to comply. You either adapt to the new sizes, or you pay the cost (time and money) to change providers (see [Appendix B](#) about lock-in).

Most importantly, the fundamental infrastructure you build is no longer available from a single critical server. If you adopt cloud native infrastructure, whether you like it or not, you're building a distributed system.

⁵ Fundamentals of Kaizen and Agile development.

The old practices to keep services available by simply avoiding failure do not work. The goal is no longer the maximum number of nines you can engineer for—it is the minimum number of nines you can get away with.

Site Reliability Engineering explains it this way:

It's both unrealistic and undesirable to insist that SLOs will be met 100% of the time: doing so can reduce the rate of innovation and deployment, require expensive, overly conservative solutions, or both. Instead, it is better to allow an error budget—a rate at which the SLOs can be missed—and track that on a daily or weekly basis.

The goal of engineering is not availability; it's creating business value. You should make resilient systems but not at the expense of overengineering a solution to avoid chaos.

The old methods of testing changes to prevent downtimes won't work either. It is nontrivial to create testing environments for large distributed systems. This is especially true when services are often updated and deployments are self-service.

It's impossible to take a snapshot of an environment when there are **4,000 deploys per day at Netflix** or **10,000 concurrently running versions of Facebook**. Testing environments need to be dynamically segregated portions of production. The infrastructure needs to support this method of testing and embrace the failures that will come from frequently testing new code in production.

You can test for some chaos (see **Chapter 5**), but chaos is not predictable by definition. Prepare your infrastructure and applications to react to chaos predictably, and don't try to avoid it.

Applications

The purpose of infrastructure is to run applications. If your business were solely based on providing infrastructure to other companies, your success would still hinge on their ability to run applications.

If you build it, they are not guaranteed to come. Any abstractions you create need to improve the ability to run applications.

Avoid “Leaky Abstractions”

Abstractions do not completely hide the implementation details of what they are abstracting. The Law of Leaky Abstractions states that “All nontrivial abstractions, to some degree, are leaky.”

What this means is the further you abstract something, the harder it is to hide the details of the thing.

For example, application resource requests usually are abstracted by requesting a percentage of a CPU core, an amount of memory, and an amount of disk storage. The physical allocation of those resources is not managed by the application directly (the API provisions them), but the request for the resources makes it obvious that the systems that run the application have those types of resources available.

If instead the abstraction were a percentage of a server or data center (e.g., 50 servers, 0.2 data centers), that abstraction does not have the same meaning because there is no such thing as a single size server or data center unit. Make sure the abstractions created are meaningful for the applications that will use them.

The benefit of cloud native practices is that as you improve your ability to run applications, you also improve your ability to run infrastructure. If you follow the patterns in Chapters 3–6, you will be well adapted to use applications to continually improve and adapt your infrastructure to the applications' needs.

Focus on building applications that are small in scope, easy to adapt, easy to operate, and resilient when failure happens. Make sure those applications are responsible for all infrastructure management and changes. If you do that, you will have created cloud native infrastructure.

Predicting the Future

If you have already adopted the patterns and practices in this book, you are now in uncharted territory. The companies that run the largest infrastructure in the world have adopted these practices. Whatever new patterns they have for running infrastructure have not been publicly disclosed or are still being discovered.

The good news is your infrastructure is now designed to be agile and change. You can more easily adapt to whatever new challenges you have.

To look at the road ahead, instead of looking at existing infrastructure patterns, we can look at where infrastructure gets its inspiration—software. Almost every pattern that has been adopted by infrastructure came from patterns in software development.

For an example, look at distributed applications. Many years ago, when software was exposed to the internet, running the application on a single server under a single codebase would not scale. This included performance and process limitations for managing the applications.

The application needed to be duplicated onto additional servers and then load balanced to meet demand. When this reached limits, it was broken apart into smaller components, and APIs were created to remotely call the functionality over HTTP instead of through an application library.

Infrastructure has taken a similar approach to scaling; it just adapts slower than software in most areas. Modern infrastructure platforms such as Kubernetes break apart roles of infrastructure management into smaller components. The components can scale independently depending on their performance bottleneck (CPU, memory, and I/O) and can be iterated on quickly.

Some applications do not have the same scaling requirements and do not need to adapt to new architecture. One of the roles of an engineer is to know when, and when not, to adopt new technologies. Only those who know the limitations and bottlenecks of their stack should decide what is the right direction.

Keep an eye on what new solutions applications develop as they find new limitations. If you can understand what the limitation is and why the change was made, you will always be ahead of the curve in infrastructure.

Conclusion

Our goal with this book was to help you better understand what cloud native infrastructure is and why you would want to adopt it. While we believe it has many benefits over traditional infrastructure, we do not want you to blindly use any technology without understanding it.

Do not expect to gain all of the benefits without adopting the culture and processes that come with it. Just running infrastructure in a public cloud or running a container orchestrator will not change the process for how that infrastructure adapts.

It is easier to manually create a handful of VMs in Amazon Web Services, SSH into each one, and create a Kubernetes cluster than it is to change how people work. The former is not cloud native.

Remember, the applications that provision your infrastructure are not static snapshots in time; they should continually run and drive the infrastructure toward a desired state. Managing infrastructure is not about maintaining hosts. You need to create abstractions for resources, APIs to represent those abstractions, and applications to consume them.

The goal is to meet the needs of your applications and to write applications that are the software equivalent of your business process and job functions. The software needs to easily adapt, as processes and functions change frequently.

When you master that, you will continually find challenges in creating new abstractions, keeping services resilient, and pushing the limits of scalability. If you are able to find new limitations and useful abstractions, please give back to the communities you are a part of.

Open source and giving back through research and community is how these patterns have emerged from the environments that pioneered them. Sharing allows more innovation and insights to spread and makes the longevity and adaptability of these practices easier on everyone.

Innovation doesn't come only from finding solutions or building the next great product. It takes the seemingly unimportant form of asking questions, speaking up, and failing.

Please keep doing all those things, especially failing and sharing.

Patterns for Network Resiliency

Applications need to be resilient when running in a cloud environment. One important area especially prone to failure is network communications. One common pattern for adding network resiliency is to create a library that is imported into applications, which provides the network resiliency patterns described in this appendix. However, imported libraries become difficult to maintain for services written in many languages, and when new versions of the network library are released, it puts an additional burden on applications to test and redeploy.

Instead of making applications handle network resiliency logic, it is possible to put a proxy in place that can act as a layer of protection and enhancement for applications. A proxy has the advantage of sheltering the applications from needing additional complex code and minimizing developer effort for initial and ongoing development.



Network resiliency logic can be handled in the connection layer (physical or SDN), in the application, or via a transparent proxy. While proxies are not part of the traditional network stack, they can be used to transparently manage network resiliency for the applications.

Transparent proxies can run anywhere in the infrastructure, but are more beneficial the closer they are to the applications. They also need to be as comprehensive as possible in protocols and what **Open Systems Interconnection model (OSI model) layers** they can proxy.

Proxies play an active role in infrastructure resiliency by implementing the following patterns:

- Load balancing
- Load shedding
- Service discovery
- Retries and deadlines
- Circuit breaking

Proxies can also be used to add functionality to applications. Some features include:

- Security and authentication
- Routing (ingress and egress)
- Insight and monitoring

Load Balancing

There are many ways to load balance an application and plenty of reasons why you should always put a load balancer in front of cloud native applications:

DigitalOcean explains some good reasons in [“5 DigitalOcean Load Balancer Use Cases”](#):

- Horizontal scaling
- High availability
- Application deployments
- Dynamic traffic routing

Coordinated transparent proxies (e.g., envoy and linkerd) are one way to load balance applications. Some benefits for having a transparent proxy handle load balancing are:

- A view of requests for all endpoints allows for better load-balancing decisions.
- Software-based load balancers allow flexibility in picking the right way to balance load.

Transparent proxies don't have to blindly pass traffic to the next router. As Buoyant points out in their blog post [“Beyond Round Robin: Load Balancing for Latency,”](#) they can be centrally coordinated to have a broader view of the infrastructure. This allows the load balancing to globally optimize traffic routing instead of only locally optimizing for quick packet handoffs.

With more knowledge about endpoints and which services are making and receiving requests, the proxy can be more rational about where to send traffic.

Load Shedding

The *Site Reliability Engineering* book explains that load shedding is not the same as load balancing. While load balancing tries to find the correct backend to send traffic, load shedding intentionally drops traffic if the application cannot take the request.¹

By dropping load to protect the application instance, you can ensure the applications don't restart or are forced into unfavorable conditions. Dropping a request is much faster than waiting for timeouts and requiring applications to restart.

Load shedding can help protect application instances when something is broken or there is too much traffic. When things are working properly, the application should discover other dependent services with service discovery.

Service Discovery

Service discovery is usually handled by the orchestration system running the services. Transparent proxies can tie into the same data and provide additional features.

Proxies can enhance standard service discovery by tying multiple sources together (e.g., DNS and a key-value database) and presenting them in a uniform interface. This allows implementors to change their backend without rewriting all application code. If service discovery is handled outside of the application, it may be possible to change the service discovery tooling without rewriting any application code.

Because the proxy can have a more holistic view of requests in the infrastructure, it can decide when endpoints are healthy or not. This works in conjunction with other features, such as load balancing and retries to get traffic routed to the best endpoint.

Proxies can also take additional metadata into account when allowing services to discover each other. They can implement logic, such as node latency or “distance,” to make sure the correct service is discovered for a request.

Retries and Deadlines

Typically an application would use built-in logic to know what to do with a failed request to an external service. This can also be handled by a proxy without additional application code.

¹ This may seem like a bad idea; retries help make sure calls don't completely fail.

Proxies intercept all ingress and egress traffic to an application and route the requests. If an outgoing request comes back as failed, the proxy can automatically retry without needing to involve the application. If the request returns for any other reason, the proxy can process appropriately, based on rules in its configuration.

This is great, so long as the application is resilient to latency. If not, the proxy should return the failure notice based on an application's deadline.

Deadlines allow applications to specify how long a request is allowed to take. Because the proxy can “follow” requests to their destination and back, it can enforce deadline policies across all applications that use the proxy.

When a deadline has been exceeded, the failure is returned to the application, and it can decide the appropriate action. An option may be to degrade the service, but the application may also choose to send an error back to the user.

Circuit Breaking

The pattern is named for the same circuit breakers in home wiring. Circuits default to a “closed” state when everything is working normally, and allow traffic to flow through the breaker. When failures are detected, the circuit “opens” and breaks the flow of traffic.

The Retry Pattern enables an application to retry an operation in the expectation that it will succeed. The Circuit Breaker pattern prevents an application from performing an operation that is likely to fail.

—Alex Homer, *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications*

A broken circuit can be to an individual endpoint or an entire service. Once it's open, no traffic will be sent, and all attempts to send traffic will immediately return as failed.

Unlike home circuits, even in an open state the proxy will test the failed endpoint. Failed endpoints can be put in a “half open” state when detected to be available again after a failure. This state will send small amounts of traffic until the endpoint is marked as failed or healthy.

This pattern can make applications much faster by failing fast and routing only to healthy endpoints. By continually checking endpoints, the network can self-heal and intelligently route traffic.

In addition to these resiliency features, proxies can enhance applications in the following ways.

TLS and Auth

Proxies can terminate transport layer security (TLS) or any other security supported by the proxy. This allows for the security logic to be centrally managed and not re-implemented in every application. New security protocols or certificates can then be updated across the infrastructure without applications needing to be redeployed.

The same is true for authentication. However, authorization should still be managed by the application because it is usually a more fine-grained, application-specific feature. User session cookies can be validated by the proxy before the application ever sees them. This means only authenticated traffic will be seen by the application.

This will not only save time for the application, but can prevent downtimes through certain types of abuses as well.

Routing (Ingress and Egress)

When proxies run in front of all your applications, they control traffic going in and out of the application. They can also manage the traffic going in and out of the cluster.

Just as reverse proxies can be used to route to backends in an *N*-tier architecture, the service proxies can be exposed to traffic outside the cluster and used to route requests coming in. This is another datapoint the proxies can use to know where traffic is coming from and where it is being sent.

Having a reverse proxy with insight into all the service-to-service communication means route choices can be better informed than traditional reverse proxies.

Insight and Monitoring

With all of this knowledge about traffic flow inside the infrastructure, the proxy system can expose metrics about individual endpoints and cluster-wide views of traffic. These datapoints have traditionally been exposed in proprietary network systems or difficult-to-automate protocols (e.g., SNMP).

Because the proxies know immediately when endpoints are unreachable, they are the first to know when the endpoint is unhealthy. The orchestration system can also check application health, but the application may not know it is unhealthy to report the correct status to the orchestration tool. With knowledge about all endpoints serving the same service, proxies can also be the best place to monitor service health.

Lock-in

There is much debate about using a cloud provider and avoiding vendor lock-in. The debate is often more ideological than practical.

Lock-in is often a concern for engineers and management. It should be weighed as a risk to an application in the same way as choosing a programming language or framework would be. The selection of a programming language and cloud provider are forms of lock-in, and it is the engineer's responsibility to understand the risks and know when the risk is acceptable.

There are a few things to keep in mind when you are choosing a vendor or technology:

- Lock-in is unavoidable.
- Lock-in is a risk, but not always high.
- Don't outsource thinking.

Lock-in Is Unavoidable

In technology there are two types of lock-in:

Technology lock-in

Deals with decisions lower in the stack of development

Vendor lock-in

Most often, deals with services and software that are consumed as part of the project (vendor lock-in can also include hardware and operating systems, but we will focus only on services)

Technology Lock-in

Developers will choose technologies they are familiar with or ones that provide the greatest benefit for the application being developed. These technologies can range from vendor-provided technologies (e.g., .NET and Oracle Database) to open source software (e.g., Python and PostgreSQL).

The lock-in provided at this level often requires conforming to an API or specification that will influence how the application is developed. There are sometimes alternatives to chosen technologies, but they often come with a high switching cost, because the technology influences much about how the application was designed.

Vendor Lock-in

Vendors, such as cloud providers, are a different form of lock-in. In this case, you are consuming resources from the vendor. This can be infrastructure resources (e.g., compute and storage), or it can be hosted software (e.g., Gmail).

The higher up the stack you consume resources, the more value you should get from the resource being consumed (e.g., Heroku). High-level resources are the most abstracted away from lower-level resources and allow products to be productive faster.

Lock-in Is a Risk

Technology lock-in is often a one-time decision or agreement with a vendor to use the technology. If you no longer have a support agreement with the vendor, your software does not immediately break; it just becomes self-supported.

Open source software can reduce the amount of lock-in from technology, but it does not eliminate it. Using open standards reduces the lock-in even more, but it's important to know the difference between open standards and open source.

Just because someone else wrote the code doesn't make it a standard. Likewise, proprietary systems can form unofficial standards that allow choice to migrate away from them (e.g., AWS S3).

The reason vendor lock-in is often spoken about more than technology lock-in is because vendor lock-in has a higher risk than technology. If you don't pay the vendor, your application will cease to run; you no longer have access to the resources you were paying for.

As alluded to earlier, vendor services provide more value because they allow product development to not require all of the lower-level implementations. Don't avoid hosted services to eliminate risk; you should weigh the risk and reward of the service just as you would anything else.

If the service provides a standard interface, it is very low risk. The more custom the interface is, or the more unique the product is, the higher the risk it will be to switch.

Don't Outsource Thinking

One of the goals of this book is to help you make decisions yourself. Don't blindly take advice from other people or reports without knowing the context of the advice and how it applies to your situation.

If you can deliver products faster by consuming hosted cloud services, you should pick a vendor and start consuming them. While measuring risk is good, having endless debates over multiple vendors with similar solutions and building the service yourself is not a good use of time.

If multiple vendors provide similar services, pick the one that is easiest to adopt. Limitations will quickly surface after you begin using the service. The biggest factor when picking a vendor is to choose one with the same pace of innovation as you.

If the vendor innovates faster than you, then you will not be able to take advantage of its newest technologies and may have to spend much of your time migrating from older technologies. If the vendor's innovation is too slow, then you will have to build your own abstractions on top of what the vendor provides, and you will not be focusing on your business objectives.

To stay competitive, you may need to consume resources that do not yet have standards or alternatives (e.g., new and experimental services). Don't be scared of them just because they may leave you locked in to that service. Weigh the risk of staying competitive or losing market share to your competition, who may innovate faster.

Understand what risks you are not able to avoid and what risks are too much for your business. Make decisions to maximize reward and minimize risk.

Box: Case Study



The following was originally published on [Kubernetes.io](https://kubernetes.io) by the CNCF and is used here with permission.

In the summer of 2014, Box was feeling the pain of a decade’s worth of hardware and software infrastructure that wasn’t keeping up with the company’s needs.

A platform that allows its more than 50 million users (including governments and big businesses like General Electric) to manage and share content in the cloud, Box was originally a PHP monolith of millions of lines of code built with bare metal inside of its own data centers. It had already begun to slowly chip away at the monolith, decomposing it into microservices. “And as we were expanding into regions around the globe, the public cloud wars were heating up and we started to focus on how to run our workload across many different environments and many different cloud infrastructure providers,” says Box cofounder and services architect Sam Ghods. “It’s been a huge challenge thus far because all these different providers, especially bare metal, have very different interfaces and ways in which you work with them.”

Box’s cloud native journey accelerated that June, when Ghods attended DockerCon. The company had come to the realization that it could no longer run its applications only off bare metal, and was researching containerizing with Docker, virtualizing with OpenStack, and supporting public cloud.

At that conference, Google announced the release of its Kubernetes container management system, and Ghods was won over. “We looked at a lot of different options, but Kubernetes really stood out, especially because of the incredibly strong team of Borg veterans and the vision of having a completely infrastructure-agnostic way of being able to run cloud software,” he says, referencing Google’s internal container

orchestrator Borg. “The fact that on day one it was designed to run on bare metal just as well as Google Cloud meant that we could actually migrate to it inside of our data centers, and then use those same tools and concepts to run across public cloud providers as well.”

Another plus: Ghods liked that Kubernetes has a universal set of API objects like pods, services, replica sets, and deployments, which created a consistent surface to build tooling against. “Even PaaS layers like OpenShift or Deis that build on top of Kubernetes still treat those objects as first-class principles,” he says. “We were excited about having these abstractions shared across the entire ecosystem, which would result in a lot more momentum than we saw in other potential solutions.”

Box deployed Kubernetes in a cluster in a production data center just six months later. Kubernetes was then still pre-beta, on version 0.11. They started small: the very first thing Ghods’s team ran on Kubernetes was a Box API monitor that confirms Box is up. “It was just a test service to get the whole pipeline functioning,” he says. Next came some daemons that process jobs, which are “nice and safe because if they experienced any interruptions, we wouldn’t fail synchronous incoming requests from customers.”

The first live service, which the team could route to and ask for information, was launched a few months later. At that point, Ghods says, “We were comfortable with the stability of the Kubernetes cluster. We started to port some services over, then we would increase the cluster size and port a few more, and that’s ended up to about 100 servers in each data center that are dedicated purely to Kubernetes. And that’s going to be expanding a lot over the next 12 months, to hundreds, then thousands.”

While observing teams who began to use Kubernetes for their microservices, “we immediately saw an uptick in the number of microservices being released,” Ghods notes. “There was clearly a pent-up demand for a better way of building software through microservices, and the increase in agility helped our developers be more productive and make better architectural choices.”

Ghods reflects that as early adopters, Box had a different journey from what companies experience now. “We were definitely lock step with waiting for certain things to stabilize or features to get released,” he says. “In the early days we were doing a lot of contributions [to components such as kubectl apply] and waiting for Kubernetes to release each of them, and then we’d upgrade, contribute more, and go back and forth several times. The entire project took about eighteen months from our first real deployment on Kubernetes to having general availability. If we did that exact same thing today, it would probably be less than six.”

In any case, Box didn’t have to make too many modifications to Kubernetes for it to work for the company. “The vast majority of the work our team has done to implement Kubernetes at Box has been making it work inside of our existing (and often

legacy) infrastructure,” says Ghods, “such as upgrading our base operating system from RHEL6 to RHEL7 or integrating it into Nagios, our monitoring infrastructure. But overall Kubernetes has been remarkably flexible with fitting into many of our constraints, and we’ve been running it very successfully on our bare metal infrastructure.”

Perhaps the bigger challenge for Box was a cultural one. “Kubernetes, and cloud native in general, represents a pretty big paradigm shift, and it’s not very incremental,” Ghods says. “We’re essentially making this pitch that Kubernetes is going to solve everything because it does things the right way and everything is just suddenly better. But it’s important to keep in mind that it’s not nearly as proven as many other solutions out there. You can’t say how long this or that company took to do it because there just aren’t that many yet. Our team had to really fight for resources because our project was a bit of a moonshot.”

Having learned from experience, Ghods offers these two pieces of advice for companies going through similar challenges:

1. Deliver early and often. Service discovery was a huge problem for Box, and the team had to decide whether to build an interim solution or wait for Kubernetes to natively satisfy Box’s unique requirements. After much debate, “we just started focusing on delivering something that works, and then dealing with potentially migrating to a more native solution later,” Ghods says. “The above-all-else target for the team should always be to serve real production use cases on the infrastructure, no matter how trivial. This helps keep the momentum going both for the team itself and for the organizational perception of the project.”
2. Keep an open mind about what your company has to abstract away from developers and what it doesn’t. Early on, the team built an abstraction on top of Dockerfiles to help ensure that all container images had the right security updates. This turned out to be superfluous work, since container images are immutable and you can instead scan them post-build to ensure they do not contain vulnerabilities. Because managing infrastructure through containerization is such a discontinuous leap, it’s better to start by working directly with the native tools and learning their unique advantages and caveats. An abstraction should be built only after a practical need for it arises.

In the end, the impact has been powerful. “Before Kubernetes,” Ghods says, “our infrastructure was so antiquated it was taking us over six months to deploy a new microservice. Now a new microservice takes less than five days to deploy. And we’re working on getting it to less than a day. Granted, much of that six months was due to how broken our systems were, but bare metal is intrinsically a difficult platform to support unless you have a system like Kubernetes to help manage it.”

By Ghods's estimate, Box is still several years away from his goal of being a 90-plus percent Kubernetes shop. "So far we've accomplished having a stable, mission-critical Kubernetes deployment that provides a lot of value," he says. "Right now about 10 percent of all of our compute runs on Kubernetes, and I think in the next year we'll likely get to over half. We're working hard on enabling all stateless service use cases, and plan to shift our focus to stateful services after that."

In fact, that's what he envisions across the industry: Ghods predicts that Kubernetes has the opportunity to be the new cloud platform. Kubernetes provides an API consistent across different cloud platforms including bare metal, and "I don't think people have seen the full potential of what's possible when you can program against one single interface," he says. "The same way AWS changed infrastructure so that you don't have to think about servers or cabinets or networking equipment anymore, Kubernetes enables you to focus exclusively on the software that you're running, which is pretty exciting. That's the vision."

Ghods points to projects that are already in development or recently released for Kubernetes as a cloud platform: cluster federation, the Dashboard UI, and CoreOS's etcd operator. "I honestly believe it's the most exciting thing I've seen in cloud infrastructure," he says, "because it's a never-before-seen level of automation and intelligence surrounding infrastructure that is portable and agnostic to every infrastructure platform."

Box, with its early decision to use bare metal, embarked on its Kubernetes journey out of necessity. But Ghods says that even if companies don't have to be agnostic about cloud providers today, Kubernetes may soon become the industry standard, as more and more tooling and extensions are built around the API.

"The same way it doesn't make sense to deviate from Linux because it's such a standard," Ghods says, "I think Kubernetes is going down the same path. It's still early days—the documentation still needs work and the user experience for writing and publishing specs to the Kubernetes clusters is still rough. When you're on the cutting edge you can expect to bleed a little. But the bottom line is, this is where the industry is going. Three to five years from now it's really going to be shocking if you run your infrastructure any other way."

A

abstractions, [6](#), [21-23](#), [118](#)
activity tests, [108-109](#)
aggregation of metrics, [99](#)
agility, [9](#), [10](#), [24](#), [115](#), [119](#)
Amazon two-pizza teams, [113](#)
Amazon Web Services (AWS), [5](#)
API for infrastructure (see data structure for infrastructure)
API-driven infrastructure, [29](#), [37](#)
(see also IaaS)
application isolation, [94](#)
applications (see cloud native applications; infrastructure applications)
assertions, testing, [73-76](#)
atomicity, [41](#)
audience role in infrastructure representation, [30](#), [38](#)
audited API (see state in infrastructure application)
auditing infrastructure
 component-level (see reconciler pattern)
 high-level, [109-111](#)
authentication, [127](#)
author role in infrastructure representation, [30](#), [38](#)
automation, [17](#)
autonomous application management, [17](#)
autoscaling, [20](#), [64](#)
availability, calculating, [15](#)
AWS (Amazon Web Services), [5](#)
Azure Cloud Services, [5](#)

B

Beyer, Betsy (Site Reliability Engineering), [12](#), [14](#), [39](#), [78](#), [118](#), [125](#)
Beyond the Twelve Factor App (Hoffman), [9](#)
books
 Beyond the Twelve Factor App (Hoffman), [9](#)
 Monitoring Distributed Systems (Eweschuk), [84](#)
 Site Reliability Engineering (Beyer et al.), [12](#), [14](#), [39](#), [78](#), [118](#), [125](#)
bootstrapping problem, [44](#)
Borg, [11](#), [96](#)
Box case study, [133-136](#)
business, readiness to adopt cloud native infrastructure, [23-24](#), [26-27](#)

C

chaos testing, [78-84](#), [117-118](#)
circuit breaking pattern, [126-127](#)
cloud native applications, [9-17](#)
 availability of, calculating, [15](#)
 characteristics of, [9-17](#), [118-119](#)
 communication model for, [16](#)
 configuration for, [92](#)
 criteria for, [20](#), [24-26](#)
 debugging, [91](#), [100](#)
 deploy stage, [90](#), [104-106](#)
 designing, [87-88](#)
 health reporting in, [10-11](#)
 infrastructure management, [17](#), [20](#), [93-100](#)
 isolation for, [94](#), [96](#)
 life cycle of, [90-93](#), [104](#), [108](#)
 logging, [98](#)
 metrics aggregation for, [99](#)

- microservices implementing, 10
 - migrating to cloud native infrastructure, 115-116
 - monitoring, 98
 - resiliency of, 13
 - resource allocation, 95
 - retire stage, 93, 108-109
 - run stage, 91-93, 106-108
 - scheduling, 95
 - securing (see security)
 - service discovery for, 97
 - state management for, 97-98
 - telemetry data for, 12-13
 - tracing, 100
 - Cloud Native Computing Foundation (CNCF), viii, x, 9
 - cloud native infrastructure, viii, 1, 6-7, 113-119
 - adopting, 114-121
 - application readiness for, 20, 24-26
 - applications managed by, 17, 20, 93-100
 - benefits of, 2
 - business readiness for, 23-24, 26-27
 - deployment of (see deployment of infrastructure)
 - future of, 119
 - implementation of (see infrastructure applications)
 - incomplete implementations of, 7-8
 - layers of, 25
 - monitoring, 84
 - mutating, 63-66
 - patterns for (see patterns)
 - people's roles in, 2, 21
 - representation of (see representation of infrastructure)
 - system abstractions for, 21-23
 - testing (see testing infrastructure)
 - when not to adopt, 24-27
 - when to adopt, 19-24
 - cloud provider
 - IaaS, 4, 7, 25, 29, 37
 - inventory APIs from, 110
 - maintaining compatibility with, 65
 - SLA with, 15
 - vendor lock-in with, 130
 - CMDB (configuration management database), 110
 - CNCF (Cloud Native Computing Foundation), viii, x, 9
 - code coverage, 72
 - code representation of infrastructure, 34-36, 43
 - communication models, 16
 - compliance tests, 107-108
 - confidence levels with infrastructure, 72-73 (see also testing infrastructure)
 - configuration for applications, 92
 - configuration management, 8, 34, 39, 56
 - configuration management database (CMDB), 110
 - conformity tests, 106-107
 - consistency, eventual, 41
 - contact information for this book, xii
 - container orchestrators, 7-8, 24, 116
 - containers, 7, 95
 - controller, 57
 - conventions used in this book, xi
 - Conway's Law, 23
 - CoreOS, 63
- ## D
- data store for state (see state store)
 - data structure (API) for infrastructure, 45 (see also representation of infrastructure)
 - adding features to, 60-61
 - consuming and producing with operators, 63
 - deprecating features, 61-63
 - designing, 59-60
 - for reconciler pattern, 50-51
 - updating for infrastructure mutations, 51
 - debugging applications, 91, 100
 - declarative communication, 16
 - degradation, graceful, 14
 - deploy stage of applications, 90, 104-106
 - deployment gating, 104-106
 - deployment of infrastructure
 - from code, 36
 - from diagram, 32
 - idempotency with, 40
 - resiliency with, 40-42
 - roles in, 38
 - from script, 33
 - from software, 37
 - tools for, 38-42
 - Destroy() method, reconciler, 55
 - developers (see people)
 - DevOps, x, 2, 23
 - diagram representation of infrastructure, 30-32

DigitalOcean, 124
DSL (domain-specific language), 35
duration metric, 12
dynamic application management, 20
dynamic configuration, 92
dynamic infrastructure (see cloud native infrastructure)

E

engineers (see people)
environment isolation, 96
envoy proxy, 89, 110, 124
errors metric, 12
etcd, 11
eventual consistency, 41
Ewaschuk, Rob (Monitoring Distributed Systems), 84

F

FaaS (Function as a Service), 94
Facebook Gatekeeper, 92
failure, designing for (see resiliency)
filesystem, as state store, 46-49
fonts used in this book, xi

G

gating (see deployment gating)
GetActual() method, reconciler, 54, 57
GetExpected() method, reconciler, 54, 57
Google App Engine, 5
Google Borg, 11, 96
Google DiRT (Disaster Recovery Training), 78
graceful degradation, 14, 125
graphs, for resource maps, 52-53

H

happy tests, 70
health reporting, 10-11, 89
Heroku's 12 factors, x, 5, 9, 115
Hoffman, Kevin (Beyond the Twelve Factor App), 9
hypervisor, 3

I

IaaS (Infrastructure as a Service), 4, 7, 25, 29, 37
idempotency, 40
immutable data structure, 50
immutable infrastructure, 111

infrastructure, vii-viii, 1
(see also cloud native infrastructure)
history of, 3-6
scaling, 2, 103
infrastructure applications
API for, 45
bootstrapping problem, 44
designing, 43-58
developing, 59-66
mutating infrastructure using, 63-66
reconciler pattern for, 49-58
state, actual versus expected, 49, 54
state, as audited API, 45
state, reconciling, 53, 54, 56-58
state, storage medium for, 46-49
infrastructure as code, 34-36, 43
infrastructure as a diagram, 30-32
infrastructure as a script, 32-34
Infrastructure as a Service (IaaS), 4, 7, 25, 29, 37
infrastructure as software, 36-38, 43
infrastructure assertions, 73-76
input validation, 69-70
integration tests, 76
isolation
application isolation, 94
environment isolation, 96

K

KPI (key performance indicator), 12
Kubernetes, 22, 24, 57, 63, 133-136

L

landscape project, CNCF, x
life cycle of applications, 90-93, 104, 108
linkerd proxy, 110, 124
load balancing pattern, 124
load shedding pattern, 125
load, excessive, 14
lock-in, 2, 129-131
logging, 98

M

maintenance costs, 25
mean time between failures (MTBF), 14
mean time to recover (MTTR), 14
metrics (see telemetry data)
microservices, 8, 10, 89
Microsoft Azure, 5

mock testing, 77-78
Monitorama conference, 84
monitoring applications, 98, 127
 (see also telemetry data)
 health reporting, 10-11, 89
 SLOs, 12, 118
Monitoring Distributed Systems (Ewaschuk), 84
monitoring infrastructure, 84
monoliths, 10
MTBF (mean time between failures), 14
MTTR (mean time to recover), 14
multitenancy, 94

N

Netflix Archaius, 92
Netflix culture deck, 113
Netflix Janitor Monkey, 93, 108
Netflix OSS, 89
Netflix Simian Army, 80, 107
network resiliency, patterns for, 123-127
network traffic, tracking, 110

O

observability, 9, 91
online resources, x, xii
 DigitalOcean, 124
 Google Borg paper, 11, 96
 health checks, 11
 Heroku's 12 factors, x
 Monitorama conference, 84
 Netflix Simian Army, 107
 Open Compute Project, 22
 OpenTracing, 91
Open Compute Project, 22
OpenTracing, 91
operability, 9, 91
operators, 63
orchestrators, 7-8, 24, 116
 (see also schedulers)

P

PaaS (Platform as a Service), 4-6
patterns, viii
 benefits of, 2
 circuit breaking pattern, 126-127
 future uses of, 119
 implementing, 89

load balancing pattern, 124
load shedding pattern, 125
for network resiliency, 123-127
reconciler pattern, 49-58
retries and deadlines pattern, 125
service discovery pattern, 125
sidecar pattern, 89

people

roles in cloud native infrastructure, 2, 21, 43
roles in infrastructure representation, 30, 38
scaling, 23, 103
transitioning to cloud native practices, 114-115
platform API (see API-driven infrastructure)
Platform as a Service (PaaS), 4-6
policy, security (see security)
proprietary input, 70-71

R

rate metric, 12
reactive communication, 16
Reconcile() method, reconciler, 54, 57
reconciler pattern, 49-58
 data structure for, 50-51
 methods for, 54-56
 reconciling, as auditing, 56-58
 reconciling, guarantee from, 53
 resource map for, 52-53
 testable code created by, 69
RED method for metrics, 12
registrator, 89
repeatability (see idempotency)
reporting (see health reporting; telemetry data)
representation of infrastructure, 29-38
 (see also data structure (API) for infrastructure)
code for, 8, 34-36, 43
diagram for, 30-32
predictable interpretation of, 30
roles in, 30, 38
script for, 32-34
software for, 36-38, 43
resiliency, 9, 13-15, 40-42, 123-127
resource allocation, 95
resource map, 52-53
resources (see books; online resources)
retire stage of applications, 93, 108-109
retries and deadlines pattern, 125
routing, 127

run stage of applications, 91-93, 106-108
ruok command, Zookeeper, 11

S

SaaS (Services as a Platform), 2
sad tests, 70
scaling
 autoscaling, 20, 64
 dynamic, 20
 infrastructure, 2, 103
 people, 23, 103
schedulers, 8, 95-96
 (see also orchestrators)
script representation of infrastructure, 32-34
security, 103-112
 activity tests, 108-109
 auditing infrastructure, 109-111
 authentication, 127
 compliance tests, 107-108
 conformity tests, 106-107
 deployment gating, 104-106
 immutable infrastructure for, 111
 policy for, applying to life cycle stages, 104, 108
 policy for, as code, 103-104
 of state store, 46
 TLS, 127
self-awareness of confidence levels, 72-73
serverless platforms, 16
servers, 3
service discovery, 97
service discovery pattern, 125
service-level agreement (SLA), 15
service-level indicator (SLI), 12
service-level objective (SLO), 12, 118
services, 2
 (see also IaaS; microservices)
Services as a Platform (SaaS), 2
sets, for resource maps, 52
sidecar pattern, 89
Site Reliability Engineering (Beyer et al.), 12, 14, 39, 78, 118, 125
SLA (service-level agreement), 15
SLI (service-level indicator), 12
SLO (service-level objective), 12, 118
software representation of infrastructure, 36-38, 43
state (audited API) in infrastructure application, 45

 actual versus expected, 49, 54
 reconciling, 53, 54, 56-58
 storage medium for, 46-49
state management for applications, 97-98
state store, 46-49
system abstractions, 6, 21-23

T

technology infrastructure (see infrastructure)
technology lock-in, 130-131
telemetry data, 12-13, 99
Terraform, 34-36
test-driven development, 68
testing infrastructure, 67-84
 activity tests, 108-109
 chaos testing, 78-84, 117-118
 compliance tests, 107-108
 confidence levels with, 72-73
 conformity tests, 106-107
 entering codebase at any point, 70-71
 goals of, 67-68
 happy and sad tests, 70
 infrastructure assertions, 73-76
 input validation, 69-70
 integration tests, 76
 mock testing, 77-78
 reconciler pattern with, 69
 unit tests, 76
 what to test, 68
TLS (transport layer security), 127
tracing applications, 100
12-factor applications, x, 5, 9, 115
Twitter Finagle, 89

U

unit tests, 76

V

validation of input, 69-70
vendor lock-in, 2, 129-131
versioned API (see API-driven infrastructure)
virtualization, 3
VM (virtual machine), 3

Z

Zookeeper, 11

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 Heptio with an emphasis in containers, infrastructure, and Kubernetes. She is an ambassador for the Cloud Native Computing Foundation. Prior to Heptio, Kris worked as a developer advocate for Microsoft, as well as an engineer on Kubernetes in Azure. She has a deep technical background in the Go programming language, and has authored many successful tools in Go. Kris 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. She understands the grievances with running cloud native infrastructure via a distributed cloud native application. She lives in Seattle, WA, and spends her free time mountaineering.

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.