O'REILLY®

3rd Edition

Infrastructure as Code

Designing and Delivering Dynamic Systems for the Cloud Age

Free Chapter

Kief Morris

THIRD EDITION

Infrastructure as Code

Designing and Delivering Dynamic Systems for the Cloud Age

Kief Morris

O'REILLY®

Infrastructure as Code
by Kief Morris

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

Acquisitions Editor: John Devins
Development Editor: Jill Leonard
Production Editor: Beth Kelly
Copyeditor: Sharon Wilkey
Proofreader: Kim Cofer
Indexer: Potomac Indexing, LLC
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

June 2016:	First Edition
December 2020:	Second Edition
March 2025:	Third Edition

Revision History for the Third Edition

2025-03-13:	First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098150358* for release details.

978-1-098-15035-8

[LSI]

Table of Contents

1. What Is Infrastructure as Code?. 1

Infrastructure as Code 2

From the Iron Age to the Cloud Age 3

Cloud Age Approaches to Change Management 3

The Path to the Cloud Age 5

Strategic Goals and Infrastructure as Code 7

System Architecture Goals and Infrastructure as Code 10

Use Infrastructure as Code to Optimize for Change 11

Myth: Infrastructure Doesn't Change Very Often 11

Myth: We Can Build the Infrastructure First and Automate It Later 12

Myth: Speed and Quality Are Trade-Offs 13

The Four Key Metrics 16

Core Practices for Infrastructure as Code 16

Define Everything as Code 17

Continually Test and Deliver All Work in Progress 17

Build Small, Simple Pieces That Can Change Independently 17

Conclusion 18

CHAPTER 1

What Is Infrastructure as Code?

If you work in a team that builds and runs IT infrastructure, cloud and infrastructure automation tools should help you deliver more value in less time and do it more reliably. In practice, however, they drive ever-increasing size, complexity, and diversity of things to manage.

These technologies have become especially relevant over the past decade as organizations have brought digital technology deeper into the core of what they do. Previously, many leaders had treated the IT function as an unfortunate distraction that should be outsourced and ignored. But digitally sophisticated competitors, users, and staff drove more processes and products online, creating entirely new categories of services like streaming media, social media, and machine learning.

The cloud and automation have helped by making it far easier for organizations to add and change digital services. However, many teams have struggled to manage the proliferation of cloud-hosted products, applications, services, and platforms. As one of my clients told me, "Moving from the data center, where we were limited to the capacity of our hardware, to the cloud, where capacity is effectively unlimited, knocked down the walls that kept our tire fire contained."[1]

Using code to define and build infrastructure creates the opportunity to bring a wide set of tools, practices, and patterns to bear on the problem of designing and implementing systems. This book explores ways of doing this. I describe the problems that Infrastructure as Code can help with, the challenges of various approaches to using infrastructure code, and patterns and practices that have proven useful.

1 According to Wikipedia, a *tire fire* has two forms: "Fast-burning events, leading to almost immediate loss of control, and slow-burning pyrolysis which can continue for over a decade." Both of these seem relevant to digital infrastructure.

This chapter provides context for the rest of the book. It starts by defining Infrastructure as Code. Next, it discusses the trends affecting the spread and evolution of using code to manage infrastructure and how infrastructure architecture fits an organization's needs and technology strategy.

The central theme of this book is the need to build infrastructure that can continually evolve to meet changing requirements.

In the past few years, the terms "infrastructure delivery lifecycle" and "day two requirements" have gained popularity among vendors. In other words, it's not enough to build infrastructure; we need to be able to continually fix, update, upgrade, expand, reshape, and adapt it. The second half of this chapter expands on this thesis, including common myths people believe about infrastructure in general and cloud infrastructure in particular.

The context provided in this chapter shapes everything else in this book. If you've been working with infrastructure code and the cloud for a while, you'll probably skim over it, hopefully nodding your head. If you're new to them, this content may help clarify the approaches advocated throughout the rest of the book.

Infrastructure as Code

A literal definition of *Infrastructure as Code* is the practice of provisioning and managing infrastructure using code rather than command-line tools or ClickOps GUIs.

Interactive infrastructure management doesn't help us do things consistently or repeatably since we decide how to implement each change as we work. This leads to inconsistent implementations and mistakes. Chapter 2 talks about some of the principles and goals that using code helps to achieve, most of which are nearly impossible with interactive infrastructure management.

No-code automation tools typically provide a GUI to define the infrastructure we want—for example, by choosing options from drop-down menus. These tools usually have ways to save configurations that we build (for example, creating templates), which means we can build multiple instances consistently. They may also have ways to update existing infrastructure to maintain consistency over time.

However, no-code tools store the definitions for infrastructure in closed systems rather than in open files. With these systems, we can't exploit the vast ecosystem of tools for working with code, such as source control repositories, code scanning tools, automated testing, and automated delivery, to name just a few.

Chapter 4 details infrastructure coding tools and languages.

No-code and low-code infrastructure automation may have a place in building and managing infrastructure but generally work best at higher levels of abstraction, such as assembling components. The components themselves, however, are likely to be built using code. Read more about infrastructure componentization in Chapter 6.

Infrastructure as Code is about more than the mechanics of how infrastructure is defined and provisioned. It is about applying the principles, practices, and tools of software engineering to infrastructure.

This book explains how to use modern software development practices such as test-driven development (TDD), continuous integration (CI), and continuous delivery (CD) to make changing infrastructure fast and safe. It also describes how software design principles help create resilient, well-maintained infrastructure. These practices and design approaches reinforce one another. Well-designed infrastructure is easier to test and deliver. Automated testing and delivery drive simpler and cleaner designs.

From the Iron Age to the Cloud Age

Modern technologies like the cloud and virtualization, and tools to automate infrastructure, deployment, and testing can help us carry out tasks much more quickly than we can by managing physical hardware and manually typing commands or clicking on GUIs, as seen in Table 1-1. However, as many organizations have discovered, adopting these tools doesn't automatically bring visible benefits.

Table 1-1. Technology changes in the Cloud Age

	Iron Age	Cloud Age
Types of resources	Physical hardware	Virtualized resources
Provisioning	Takes days or weeks	Takes minutes or seconds
Processes	Manual (runbooks)	Automated (code)

The ability to provision new infrastructure in moments and to set up systems that provision it without direct human involvement can lead to uncontrolled sprawl. If we don't have good processes for ensuring that systems are well managed and maintained, the unbounded nature of cloud technology leads to spiraling technical debt.

Cloud Age Approaches to Change Management

Many organizations try to control the potential chaos by using older, traditional IT governance models. These models focus on throttling the speed of change, requiring decisions on implementation details before work begins, high-effort process gates, and strictly siloed responsibilities for teams.

However, these models were designed for the Iron Age, when we changed physical infrastructure manually. Changes were slow and expensive, making it difficult to correct mistakes. If a task would take two weeks, and a mistake could take a week or more to fix afterward, then it seemed reasonable to add an extra week up front in hopes of preventing a mistake from happening. With cloud technology, these processes add weeks to tasks that may take less than an hour to implement and a few minutes to correct, destroying the advantages of the technology.

Moreover, research suggests that these heavyweight processes were never very effective in preventing errors in the first place.[2] They can make things worse by dividing knowledge and accountability across silos and long periods.

Fortunately, the emergence of Cloud Age technologies has coincided with the growth of Cloud Age approaches to work, including Lean, Agile, and DevOps. These approaches encourage close collaboration across roles, short feedback loops with users, and a minimalist, quality-first approach to technical implementation. Automation fundamentally shifts thinking about change and risk, resulting in faster delivery and higher quality (see Table 1-2).

Table 1-2. Ways of working in the Iron Age and the Cloud Age

	Iron Age	Cloud Age
Cost of change	High	Low
Changes are	Risks to be minimized	Essential to improve quality
A change of plan means	Failure of planning	Success in learning and improving
Optimize to	Reduce opportunities to fail	Maximize speed of improvement
Delivery approach	Large batches, test at the end	Small changes, test continually
Architectures	Monolithic (fewer, larger moving parts)	Microservices (more, smaller parts)

This ability to make changes more quickly to improve quality starts with cloud technology, which creates the capability to provision and change infrastructure on demand. Automation gives us a way to exploit this capability. We can automate not only to deploy a change quickly, but also to validate the change for correctness, quality, and governance. And by defining changes as code, we create a detailed history that can be used to audit, troubleshoot, and reverse changes.

So another definition of *Infrastructure as Code* is a Cloud Age approach to automating cloud infrastructure in a way that embraces continual change to achieve high reliability and quality.

2 The *2019 Accelerate State of DevOps Report* specifically researched the effectiveness of governance approaches, and includes a discussion of findings on pages 48–52.

DevOps and Infrastructure as Code

People define DevOps in different ways. The fundamental idea of *DevOps* is collaboration across all the people involved in building and running software. This includes not only developers and operations people, but also testers, security specialists, architects, and even managers. There is no one way to implement DevOps.

Many people look at DevOps and notice only the technology that people use to collaborate across software delivery. All too often this leads to reducing DevOps to tooling. I've seen DevOps defined as using an application deployment tool (usually Jenkins), often with a separate DevOps team that adds an extra barrier across the software delivery path, which contradicts the meaning of the term.

DevOps is, first and foremost, about people, culture, and ways of working. Tools and practices like Infrastructure as Code are valuable to the extent that they bridge gaps and improve collaboration, but they aren't enough.

The Path to the Cloud Age

DevOps, Infrastructure as Code (the name, at least), and the cloud all emerged between 2005 and 2010. In the early years, these were largely experimental, dismissed by larger organizations that considered themselves too serious to need to change the way they approached IT. The first edition of this book, published in 2016, included arguments for why readers should consider using the cloud even for critical domains like finance and health care.

The mid-2010s could be considered the Shadow Age of IT. The cloud, DevOps, continuous delivery, and Infrastructure as Code were mainly used by startups or skunkworks digital departments of larger organizations. These departments were usually set up outside the remit of the existing organization, partly to protect them from the main organization's cultural norms and formal policies, which people sometimes call "antibodies." In some cases, they were used quietly within existing departments without involving the IT department, as shadow IT.

The mantra of the Shadow Age was "move fast and break things."[3] People saw casting aside the shackles of Iron Age governance as the key to explosive growth. In the view of digital hipsters, it was time to leave the crusty old-timers to their change advisory board (CAB) meetings, mainframes, and bankruptcies ("Say hello to Blockbuster and Kodak!").

3 Meta (then Facebook) CEO Mark Zuckerberg said, "Unless you are breaking stuff, you are not moving fast enough."

Cavalier attitudes toward governance made it easier for traditionalists to dismiss the newer technologies and related ideas as irresponsible and doomed to failure. At the same time, new technology enthusiasts have often ignored the real concerns and risks underpinning what may seem like legacy mindsets.

We need to learn how to leverage newer technologies and ways of working to address fundamental issues rather than either rejecting the new ways or dismissing the issues as legacy. The progression of cloud adoption is shown in Figure 1-1.

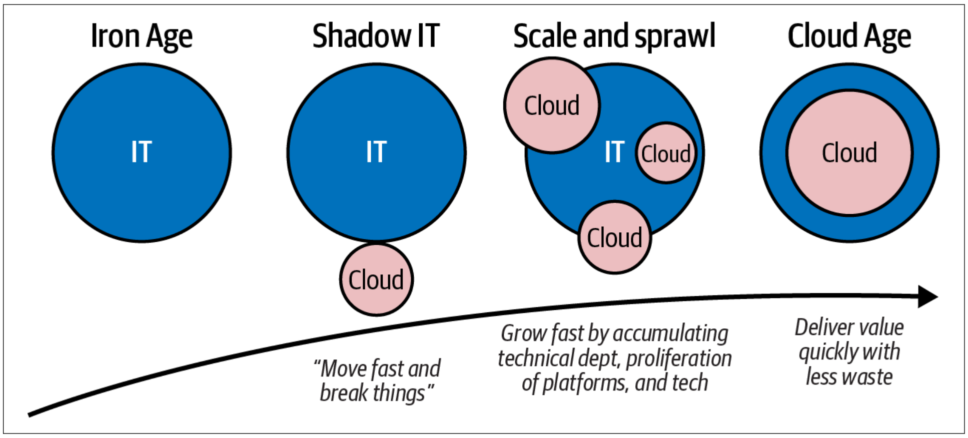
Iron Age
IT
Shadow IT
IT
Cloud
Scale and sprawl
Cloud
IT
Cloud
Cloud
Cloud Age
Cloud
"Move fast and break things"
Grow fast by accumulating technical dept, proliferation of platforms, and tech
Deliver value quickly with less waste

Figure 1-1. The path from the Iron Age to the Cloud Age

As the decade wore on and digital businesses overtook slower businesses in more and more markets, digital technologies and approaches were pulled closer to the center of even older businesses. Digital departments were assimilated, and boards asked for strategies to migrate core business systems into the cloud. This trend accelerated when the COVID-19 pandemic led to a dramatic rise in consumers and workers moving to online services. Many organizations found that their digital services were not ready for the unexpected level of demand they faced. As a result, they increased their investment and efforts in cloud technologies.

This period when cloud technology has been shifting from the periphery of business to the center can be called the Age of Sprawl. Although breaking things had gone out of fashion, moving fast was still the priority. As a result of the haste to adopt new technologies and practices, larger organizations have seen a proliferation of initiatives. Larger organizations typically have multiple, disconnected teams building platforms using various technologies, multiple cloud vendors, and varying levels of maturity and quality.

The variety of options available for building digital infrastructure and platforms[4] and the rapid pace of change within them have made it difficult to keep up to date. Platforms built on the latest technology two years ago may already be legacy.

The drivers that led to this sprawl are real. Organizations must evolve rapidly to survive and prosper in the modern digital economy. However, as I write this in late 2024, the economic landscape has changed, meaning most organizations need to be more careful with investments. Not only do we need to be choosy about which new systems and initiatives to invest in, but we also need to consider how to manage the cost of running and evolving what we already have in place. The need to grow, improve, and exploit emerging technologies has not disappeared, so the next age is not simply about cutting back and staying in place. Instead, organizations need to find sustainable ways to grow and evolve. Call it the Age of Sustainable Growth.

What does this have to do with Infrastructure as Code? Those involved in designing and building the foundational layers of our organizations' business systems must be aware of the strategic drivers those foundations must support. A key driver is rationalizing systems to sustain growth with less waste. In the years to come, our organizations' needs will shift again.

The Future Is Not Evenly Distributed

The tidy linear narrative described here as "the path to the Cloud Age" is, as with any tidy linear narrative, simplistic. Many people and organizations have experienced the trends it describes. However, none of its "ages" have ended entirely, and many drivers of different ways of thinking and working are still valid. It's essential to recognize that contexts differ. A Silicon Valley startup has different needs and constraints than a transnational financial institution. New technologies and methodologies create opportunities to handle old risks and new opportunities differently. The path to the Cloud Age is uneven and far from over. Understanding how it has unfolded so far can help us navigate what comes next.

Strategic Goals and Infrastructure as Code

Figure 1-2 shows the gap between organizational strategy and infrastructure strategy. Customer value should drive the organization's strategy, which then drives strategy to infrastructure via product and technology strategy. Each strategic layer supports the layers above it.

4 The Cloud Native Landscape diagram is a popular one for illustrating how many products, tools, and projects are available for building platforms. One of my favorite memes extends this into a CNCF conspiracy chart.

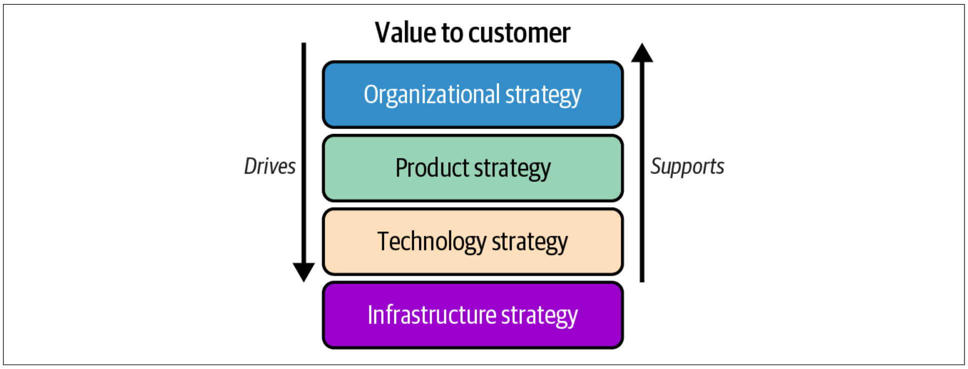

Figure 1-2. Customer value driving strategy down to infrastructure

When talking to organizations about their strategy for cloud infrastructure, I'm often struck by the gap between people responsible for the infrastructure and those responsible for organizational strategy. Engineering people are puzzled when I ask questions about the product and commercial strategy. Organizational leaders are dismissive of the need to spend time planning infrastructure capability, assuming that selecting a cloud vendor is the end of that story. Even when their infrastructure architecture creates problems with growth, stability, or security, the instinct is to demand a quick fix and move on.

The gap is not one-sided. Engineering folks tend to focus on implementing the solutions that seem obvious to them, sometimes assuming it doesn't matter what will run on it. One example of how this turns out is a company whose engineers built a multiregion cloud hosting solution with iron-clad separation between regions. The team wanted to segregate user data to avoid conflicts with various privacy regulations, so this requirement was baked deep into the architecture of their systems.

However, because neither the product nor engineering teams believed they needed close communication during development, the service was nearly ready for production rollout when it surfaced that the commercial strategy assumed that users would be able to use their accounts while traveling and working in different countries. It took considerable effort, expense, and delay to rearchitect the system to ensure that each region's privacy laws could be respected while giving users international roaming access.

So, although infrastructure can seem distant from strategic goals discussed in the boardroom, it's essential to ensure that everyone, from strategic leaders to engineering teams, understands how they are related. Table 1-3 describes a few common organizational concerns where infrastructure architecture can make a considerable difference in either enabling success or creating drag.

Table 1-3. How Infrastructure as Code is relevant to an organization's strategic goals

Business goal	Infrastructure capabilities to support	Measures of success
Deliver increasing value to customers quickly and reliably through new products and features.	Provide infrastructure needed to develop, test, and host new and existing digital services.	High performance on the four key metrics ("The Four Key Metrics" on page 16). Low effort and dependency on platform and infrastructure teams for common software delivery use cases.
Grow revenues by adding new markets, products, and customers.	Add hosting for new regions, instances of our products, and capacity.	Time to add new hosting. Incremental cost of ownership for each region, instance, product, and user.
Provide reliable, performant services to users.	Handle scaling, recovery, monitoring, and performance management services.	Availability and performance metrics.

Throughout this book, I illustrate concepts using the fictitious company FoodSpin, an online restaurant menu service. The following sidebar provides a high-level view of the company's strategy.

Introduction to FoodSpin and Its Strategy

FoodSpin is a digital service that allows restaurants to provide searchable menus for customers to order and pay for their meals. The main service is multitenancy, with multiple restaurants' menus hosted on a shared instance of the company's software. It runs separate instances in several countries, including the US, UK, Germany, and South Korea. The company also partners with large food chains, providing a dedicated, single-tenancy hosted service with customized features.

The main FoodSpin systems are Jakarta EE applications running on JBoss on virtual servers in the cloud. Some services have been rebuilt more recently as containerized applications, so their workloads are a mix of servers and containers with some serverless code.

Until recently, growth was the FoodSpin board's primary goal. The board's strategy was to spend to grow and worry about efficiency later. "Later" has arrived. The economic situation has changed, and the cost of running and developing FoodSpin's existing systems is unsustainable.

However, the company can't afford to miss opportunities to grow market share and enter new markets. So, it needs to find efficient ways to grow its footprint. An added factor is that some of the systems in place now have issues with performance and reliability, and these need to be addressed to rebuild the confidence of customers and partners.

Key organizational goals for FoodSpin include the following:

- Grow its customer base, revenue, and profits by bringing new services to market.
- Grow its customer base, revenue, and profits by expanding services to new regions.
- Retain and grow its customer base by continually improving existing services.
- Improve profitability and service quality by rationalizing its systems.

System Architecture Goals and Infrastructure as Code

An organization's strategic goals typically filter down into goals for services and technology, which can cross teams such as product development, software engineering, platform engineering, and IT operations. These groups will have their own goals, objectives, and initiatives that infrastructure architecture needs to support.

Figure 1-3 shows an example of how organizational goals, such as those described in "Introduction to FoodSpin and Its Strategy" on page 9, drive goals for an engineering organization, which in turn drive goals for the infrastructure architecture. Infrastructure as Code can ensure that environments are consistent across the path to production and multiple production instances. Read about types of environments in Chapter 12.

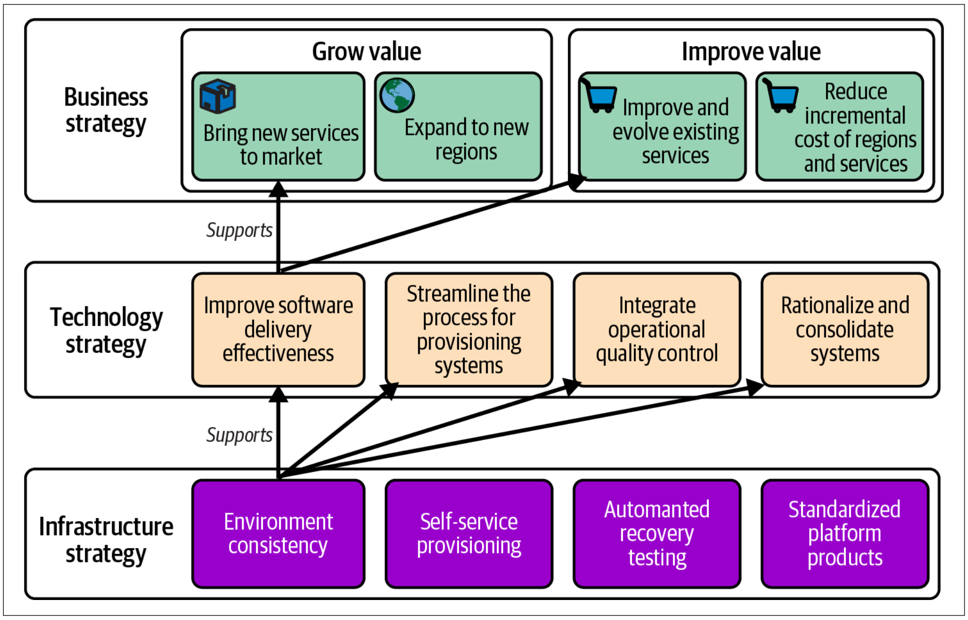
Business strategy
Grow value
Bring new services to market
Expand to new regions
Improve value
Improve and evolve existing services
Reduce incremental cost of regions and services
Supports
Technology strategy
Improve software delivery effectiveness
Streamline the process for provisioning systems
Integrate operational quality control
Rationalize and consolidate systems
Supports
Infrastructure strategy
Environment consistency
Self-service provisioning
Automanted recovery testing
Standardized platform products

Figure 1-3. Example of infrastructure goals driven by organizational goals

Consistency across environments supports the engineering goal of improving software delivery effectiveness by ensuring that test environments accurately reflect production environments. Consistency also reduces the customization needed to provision new environments for adding products or expanding into new regions.

When infrastructure is built consistently, it's easier to automate operational capabilities like security, compliance, and recovery. Less variation among environments also makes it easier to consolidate and simplify overall system architecture. So this one goal for infrastructure architecture can support multiple higher-level goals for the organization.

Use Infrastructure as Code to Optimize for Change

One of the most fundamental reasons for adopting Infrastructure as Code, although not universally understood in our industry, is to optimize the process for making changes to IT systems. If an organization fails to see benefits from adopting the cloud and automation, the most common cause is not approaching them as enablers for change.

Operations teams know that the biggest risk to a production system is changing it.[5] The Iron Age approach to managing this risk, as mentioned earlier, is to add heavyweight processes to make changes more slowly and carefully. However, adding barriers to making changes impedes fixing and improving the quality of a system.

Research from the *Accelerate State of DevOps Report* backs this up. Making changes frequently and reliably is correlated to organizational success.[6]

Rather than resisting commercial pressures to make changes frequently and quickly, modern methods of change management, from Lean to Agile, lean into the idea that this is a good thing. The ability to deliver changes both rapidly and reliably is the secret sauce for resilient, highly available, valuable systems in the digital age.

People raise several common objections when considering change as a goal for automation. These come from misunderstandings of how to use automation.

Myth: Infrastructure Doesn't Change Very Often

We want to think that we build an environment, and then it's done. In this view, we won't make many changes, so automating the process of making changes, especially testing, is a waste of time.

5 According to Gene Kim et al. in *The Visible Ops Handbook* (IT Process Institute), changes cause 80% of unplanned outages.

6 Reports from the *Accelerate* research are available in the annual *State of DevOps Report* and in the book *Accelerate* by Dr. Nicole Forsgren et al. (IT Revolution Press).

In reality, few systems stop changing before they are retired. Some people assume that a fast pace of change is temporary. Others create heavyweight change-request processes designed to discourage people from asking for changes. However, high-performing teams handle a continual stream of changes quickly and effectively. Consider these common examples of infrastructure changes:

- An essential new application feature requires a new data processing service.
- A new application feature needs the messaging service upgraded to a newer version.
- Profiling shows that the current application deployment architecture is limiting performance. We can address this by redeploying the applications across multiple clusters globally, which requires changing cloud accounts and network architecture.
- We discover a security vulnerability in our container cluster software. We must patch clusters across multiple regions and in our development and testing systems.
- The API gateway experiences intermittent failures. To diagnose and resolve the problem, we need to make a series of configuration changes.
- We find a configuration change that improves the performance of the database.

An infrastructure team with heavyweight change processes accumulates a backlog of outdated, unpatched systems, hindering the organization's ability to adapt to challenges and opportunities.

Myth: We Can Build the Infrastructure First and Automate It Later

Getting started with infrastructure automation is a steep curve. Setting up the tools, services, and working practices to automate infrastructure delivery is loads of work, especially when migrating simultaneously to a new cloud platform or technology stack. The value of this work is hard to demonstrate before starting to build and deploy services with it. Even then, the value may not be apparent to people who don't work directly with the infrastructure.

Stakeholders may pressure infrastructure teams to build new cloud-hosted systems by hand, thinking it will be quicker and can be automated later. However, automating afterward is impractical for several reasons:

- Automation can enable a new system to be delivered more quickly. Automating after the system is in place sacrifices this opportunity.
- Automation makes it easier to write automated tests for what we build. It also facilitates fixing and rebuilding when we find problems quickly. Doing this as part of the build process helps us build a more robust infrastructure.

- Automation is integral to a system's design and implementation, so adding automation to a system built without it involves significant rework.

Cloud infrastructure built without automation becomes a write-off sooner than we'd like. The cost of manually maintaining and fixing the system can escalate quickly. If the service is successful, stakeholders will prioritize expanding and adding new features over going back to add automation.

The same is true when building a system as an experiment. Once a proof of concept is up and running, people want to move on to the next thing rather than go back and build it right. And in truth, automation should be a part of the experiment. If we intend to use automation to manage our infrastructure, we need to understand how it will work, so it should be part of our proof of concept.

The solution is to build the system incrementally, automating as we go. The trick is to start with the bare minimum of automation needed to deliver the first increment of the system rather than building a complete automation system first. Starting with ready-made solutions can help, even if we intend to adopt something more sophisticated later.

For example, a team wanted to use an advanced, packaged secrets management solution but knew it would take several weeks to implement it properly. The team members chose to use the cloud platform's built-in secrets storage service initially, so the development team could start working to get the first increment of business functionality in place. They deployed the packaged solution later while the developers were working, rather than making them wait.

Myth: Speed and Quality Are Trade-Offs

It's natural to think that we can move fast only by skimping on quality and that we can get quality only by moving slowly. Many people see this as a continuum, as shown in Figure 1-4.

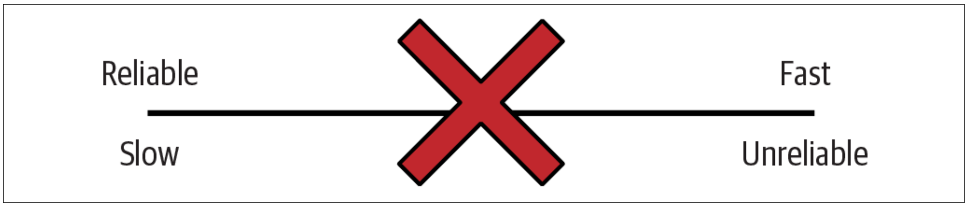

Figure 1-4. The idea that speed and quality are opposite ends of a spectrum is a false dichotomy

However, the *Accelerate* research shows otherwise:[7]

> These results demonstrate that there is no trade-off between improving performance and achieving higher levels of stability and quality. Rather, high performers do better at all of these measures. This is precisely what the Agile and Lean movements predict, but much dogma in our industry still rests on the false assumption that moving faster means trading off against other performance goals, rather than enabling and reinforcing them.
>
> —Dr. Nicole Forsgren, Accelerate

In short, organizations can't choose between being good at change or being good at stability. They tend to either be good at both or bad at both.

Here's a fundamental truth of the Cloud Age: *Stability comes from making changes*. The longer it takes to make a change, the slower we are to fix things. The flow of changes needed, like those listed previously to rebut the myth that infrastructure doesn't change often, will outpace the capacity to make them. Systems are left unpatched and with quick-fix workarounds to "known issues."

If our systems aren't fully patched, they are not stable; they are vulnerable. If we can't fix issues as soon as we discover them, the system is not stable. If we can't recover from failure quickly, the system is not stable. If making changes involves considerable downtime, the system is not stable. If changes frequently fail, the system is not stable.

Quality and speed should be seen as a quadrant rather than a continuum, as shown in Figure 1-5.

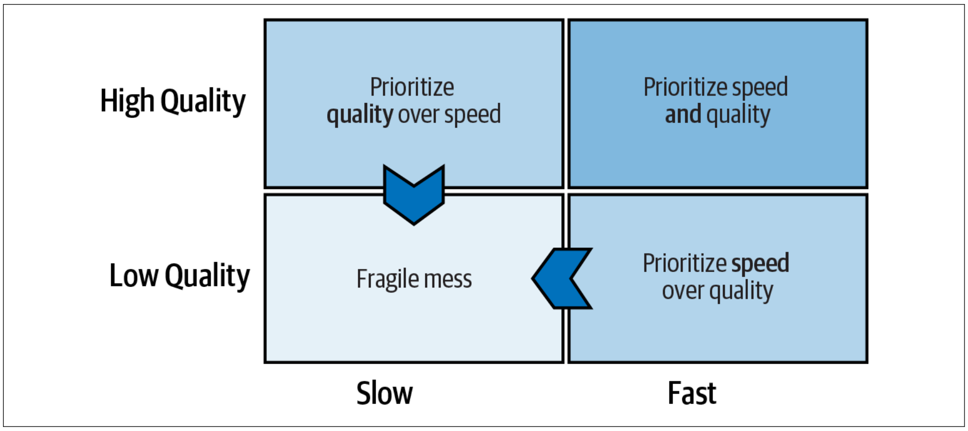
High Quality
Low Quality
Prioritize quality over speed
Prioritize speed and quality
Fragile mess
Prioritize speed over quality
Slow
Fast

Figure 1-5. Speed and quality are not trade-offs and can be combined

7 *Accelerate* by Dr. Nicole Forsgren et al. Also see the *Accelerate State of DevOps Report*.

This quadrant model shows how trying to choose between speed and quality leads to doing poorly at both:

Lower-right quadrant: prioritize speed over quality
: This is the "move fast and break things" philosophy. Teams that optimize for speed and sacrifice quality build messy, fragile systems. They slide into the lower-left quadrant because their shoddy systems slow them down. A common pattern for startups is seeing development slow after a year or two, leading founders to despair that their team has lost their mojo. Simple changes that the team would have whipped out quickly in the old days now take days or weeks because the system is a tangled mess. This is a consequence of a system built in a rush, without treating quality as a priority.

Upper-left quadrant: prioritize quality over speed
: Also known as "We're doing serious and important things, so we have to do things *properly*." Then deadline pressures drive "workarounds." Heavyweight processes create barriers to improvement, so technical debt grows along with lists of "known issues." These teams slump into the lower-left quadrant. They end up with low-quality systems because improving them is too hard. They add more processes in response to failures. These processes make it harder to make improvements and increase fragility and risk. This leads to more failures and more process. Many people working in organizations that work this way assume this is normal, especially those who work in risk-sensitive industries.[8]

The upper-right quadrant is the goal of modern approaches like Lean, Agile, and DevOps. Being able to move quickly while also maintaining a high level of quality may seem like a fantasy. However, the *Accelerate* research proves that many teams do achieve this. So this quadrant is where high performers are found.

Antifragility

Nassim Nicholas Taleb coined the term "antifragile" in his book of the same title to describe systems that actually grow stronger when stressed. Taleb's book is not IT-specific—his main focus is on financial systems—but his ideas are relevant to IT architecture.

8 This is an example of "normalization of deviance," which means people get used to working in ways that increase risk. Diane Vaughan defined this term in *The Challenger Launch Decision* (University of Chicago Press). It's ironic (and scary) that so many people in industries like finance, government, and health care consider fragile IT systems—and processes that obstruct improving them—to be normal and even desirable.

The Four Key Metrics

Navigating into the high-performing quadrant is challenging. The DevOps Research and Assessment (DORA) *Accelerate* team identifies four key metrics for software delivery and operational performance that can help keep a team on track.[9] Its research surveys various measures and has found that these four have the strongest correlation to how well an organization meets its goals:

Delivery lead time
: The elapsed time it takes to implement, test, and deliver changes to the production system

Deployment frequency
: How often changes are deployed to production systems

Change fail percentage
: The percentage of changes that either cause an impaired service or need immediate correction, such as a rollback or emergency fix

Mean time to restore (MTTR)
: The amount of time it takes to restore service after there is an unplanned outage or impairment

The research shows that organizations that perform well against their goals—whether that's revenue, share price, or other criteria—also perform well against these four metrics. The ideas in this book aim to help teams perform well on these metrics. Three core practices for Infrastructure as Code can help achieve this.

Core Practices for Infrastructure as Code

We can build and maintain highly effective systems by using Infrastructure as Code to deliver changes continually, quickly, and reliably. The various principles, practices, and techniques described throughout this book can help to achieve this. Underlying all this are a few core practices:

- Define everything as code.
- Continually test and deliver all work in progress.
- Build small, simple pieces that can be changed independently.

Each of these core practices is worth examining in more detail.

9 DORA, now part of Google, is the team behind the *Accelerate State of DevOps Report* I cited earlier.

Define Everything as Code

Defining everything "as code" is a core practice for making changes rapidly and reliably. There are a few reasons that this helps:

Reusability
: If we define a thing as code, we can create many instances of it. We can repair and rebuild things quickly, and other people can build identical instances of the thing.

Consistency
: Things built from code are built the same way every time. This makes system behavior predictable, makes testing more reliable, and enables continual testing and delivery.

Visibility
: Everyone can see how the thing is built by looking at the code. People can review the code and suggest improvements. They can learn techniques to use in other code, gain insight to use when troubleshooting, and review and audit for compliance.

Continually Test and Deliver All Work in Progress

Effective infrastructure teams are rigorous about testing. They use automation to deploy and test each component of their system and integrate all the work everyone has in progress. They test as they work rather than waiting until they've finished.

The idea is to *build quality in* rather than trying to *test quality in*.

One part of this practice that people often overlook is that it involves integrating and testing *all work in progress*. On many teams, people work on code in separate branches and integrate when they finish. According to the *Accelerate* research, however, teams get better results when everyone integrates their work at least daily. CI involves merging and testing everyone's code throughout development. CD takes this further, keeping the merged code always production-ready.

I go into more detail on how to continually test and deliver infrastructure code throughout the chapters of Part III.

Build Small, Simple Pieces That Can Change Independently

Teams struggle when their systems are large and tightly coupled. The larger a system is, the harder it is to change, and the easier it is to break.

The codebase of a high-performing team is visibly different from other codebases. The system is composed of small, simple pieces. Each piece is easy to understand and has clearly defined interfaces. The team can easily change each component on its own and can deploy and test each component in isolation.

I dig more deeply into design principles and techniques in Part II.

Conclusion

Traditional Iron Age approaches to software and system design were based on the belief that, if we are sufficiently skilled, knowledgeable, and diligent, we can come up with the correct design for the system's needs before we start working on it. In reality, we don't know the correct design until the system is already being used. Worse, changes to an organization's situation, environment, and opportunities mean the system's needs are a moving target. Even if we do find and implement the correct design, it won't remain correct for very long.

The only thing we know for sure when designing a system is that it will need to change after it's in use, not once, but continually until the system is no longer needed. The essence of Cloud Age, Lean, Agile, DevOps, and similar philosophies is designing and implementing systems so that we can continually learn and evolve our systems.

With infrastructure, this means exploiting speed to improve quality and building in quality to gain speed. Automating infrastructure takes work, especially when people are learning the tools and techniques. But doing that work helps ensure that the system can be kept relevant and useful throughout its lifespan. The next chapter discusses more specific principles for using code to design and build cloud infrastructure.

About the Author

Kief Morris (he/him) is a distinguished engineer at Thoughtworks. He drives conversations across roles, regions, and industries at companies ranging from global enterprises to early-stage startups. He enjoys working and talking with people to explore better engineering practices, architecture design principles, and delivery practices for building systems on the cloud.

Kief ran his first online system, a bulletin board system (BBS) in Florida in the early 1990s. He later enrolled in a master's degree program in computer science at the University of Tennessee because it seemed like the easiest way to get a real internet connection. Joining the CS department's system administration team gave him exposure to managing hundreds of machines running a variety of Unix flavors.

When the dot-com bubble began to inflate, Kief moved to London, drawn by the multicultural mixture of industries and people. He's still there, living with his wife, son, and cat.

Most of the companies Kief worked for before Thoughtworks were post-startups, looking to build and scale. The titles he's been given, or self-applied, include software developer, systems administrator, deputy technical director, R&D manager, hosting manager, technical lead, technical architect, consultant, and director of cloud engineering.

Colophon

The animal on the cover of *Infrastructure as Code* is Rüppell's vulture (*Gyps rueppellii*), native to East Africa and the Sahel. It is named in honor of 19th-century German explorer and zoologist Eduard Rüppell.

This large bird has a wingspan of 7–8 feet and weighs 14–20 pounds. It has mottled brown feathers and a yellowish-white neck and head. Like all vultures, this species is carnivorous and feeds almost exclusively on carrion. They use their sharp talons and beaks to rip meat from carcasses and have backward-facing spines on their tongue to thoroughly scrape bones clean.

The Rüppell's vulture is monogamous and mates for life, which can be 40–50 years long. Breeding pairs build their nests near cliffs out of sticks and lined with grass and leaves. Only one egg is laid each year—by the time the next breeding season begins, the chick is just becoming independent.

While normally silent, these are very social birds who will voice a loud squealing call at colony nesting sites or when fighting over food. This vulture does not fly very fast (about 22 mph), but it flies for 6–7 hours per day and ventures up to 90 miles from the nest in search of food. Rüppell's vultures are the highest-flying birds on record; there is evidence of one flying 37,000 feet above sea level, as high as commercial aircraft. They have a hemoglobin variant in their blood that allows them to absorb oxygen efficiently while at high altitudes.

The IUCN conservation status of Rüppell's vulture is critically endangered. Though loss of habitat is one factor, the most serious threat is poisoning. The vulture is not even the intended target: farmers often poison livestock carcasses to retaliate against predators like lions and hyenas, and hundreds of birds can be killed at a time. The total population is estimated to be 22,000 birds. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

Color illustration by Karen Montgomery, based on a black and white engraving from Cassell's *Natural History*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold 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.